Benjamin Devèze
Spécialisation SCIA

Promo 2005
UID 15015

06/15/2005

NON CLASSIFIÉ

Rapport de Stage

Sujet : *Réalisation d'une bibliothèque de déploiement de systèmes multiagents*

Préparé pour :
EPITA, Département des Stages

Préparé par :
THALES SYSTÈMES AÉROPORTÉS

THALES THALES SYSTEMES AEROPORTES	NUMERO DOCUMENT	FORMAT	PAGE
	TASFR0039964800	**A4**	**1/90**
F0052		--	REV

MAITRISE DOCUMENTAIRE

Action	*Nom*	*Date*	*Signature*
Rédigé par	Benjamin DEVEZE	20/05/2005	
Approuvé par	Patrick TAILLIBERT	13/06/2005	

Indice	*Date*	*Modification*

Table des Matières

Table des figures

Remerciements

J'adresse ma plus grande gratitude à M. Patrick Taillibert, responsable de ce stage, pour sa patience, sa disponibilité, sa confiance et sa gentillesse ainsi que pour avoir permis, par sa collaboration et sa présence, d'obtenir les résultats escomptés dans une excellente ambiance de travail.

Je remercie Mlle Caroline Chopinaud qui a été d'un grand soutien tout au long de ces six mois et qui a donné sans compter pour répondre à nos questions souvent peu pertinentes. Nous lui sommes également tous redevables des viennoiseries hebdomadaires que nous avons eu coutume de savourer tous les vendredi matins.

Merci, également, à l'ensemble des équipes de Thales pour leur accueil chaleureux et plus particulièrement à M. Bernard Botella, M. Eric Maes et M. Serge Varennes.

J'ai bien sûr une pensée toute particulière pour l'ensemble des stagiaires qui ont également contribué à l'atmosphère détendue mais malgré tout productive qui a régné durant ce stage. Je salue d'autre part leurs participations aux conversations passionnantes et passionnés qui furent échangées sur les sujets les plus divers.

Enfin, je remercie les développeurs d'OpenOffice.org ainsi que l'ensemble de la communauté Open Source pour avoir mis à la disposition de tous, les outils de travail que nous utilisons quotidiennement.

Résumé

Mon stage de fin d'études a été mené au sein de la section Aerospace Division du groupe Thales, dans le cadre d'une équipe de recherche et développement en *Intelligence Artificielle*. Mon maître de stage, M. Patrick Taillibert, investigue notamment dans le domaine des systèmes multiagents dont l'un des but est de réduire la complexité des systèmes. L'une des notions clés devant nous guider dans cette voie est l'autonomie des agents.

L'objet du stage était de développer une bibliothèque générique, nommée ALBA et écrite en Prolog, permettant de déployer des systèmes multiagents. Cette bibliothèque devait offrir au programmeur d'agents toutes les fonctionnalités dont il pourrait vouloir disposer. ALBA permet ainsi aux agents de s'envoyer des messages, de créer d'autres agents, de migrer sur des machines distantes... Un algorithme de recherche d'agent a également été implémenté. Un soin tout particulier a été porté sur la robustesse, la décentralisation, l'efficacité, la généricité et la portabilité de notre bibliothèque. Un autre point central du développement a résidé dans l'abstraction maximale des informations de bas niveau, notamment liées aux communications, tout devant être transparent pour le programmeur d'agent qui n'a pas à se soucier de ce genre de contingences matérielles à son niveau. Une bibliothèque d'évènements a également été mise en chantier afin de permettre à des outils externes d'injecter du code en des points précis d'ALBA sans retoucher à ses routines.

Les résultats sont satisfaisants puisque le coeur de la bibliothèque fonctionne très bien. Ainsi la version d'ALBA qui fut développé durant le stage a déjà eu l'occasion d'être utilisée et éprouvée dans plusieurs applications de l'équipe.

En sus du développement de cette bibliothèque, un effort relativement important a été investi dans un outil, nommé PrologDoc, permettant de générer de la documentation HTML à partir de code source Prolog dûment commenté.

Nous avons également contribué à la rationnalisation de l'organisation interne des applications et outils développés par l'équipe.

Le bilan de ce stage apparaît donc comme très positif, tant pour l'entreprise qu'à titre personnel. L'objet du sujet de stage a été réalisé dans les délais et il est déjà utilisé dans plusieurs applications internes : application de localisation passive, plate-forme de déploiement...

D'un point de vu personnel, ce stage a été riche d'enseignements, tant humains que techniques.

Chapitre 1
Introduction

1.1 Commentaires liminaires

Avant de commencer à rentrer dans le vif du sujet, il est sans doute de bon aloi de prévenir le lecteur qu'un certain nombre de termes techniques, souvent propres au systèmes multiagents ou à l'organisation de Thales, figurent ponctuellement au sein de ce rapport. C'est pourquoi, je vous renvoie directement au glossaire pour les termes et abréviations qui vous seraient inconnus. Toutefois, certains concepts seront aussi explicités dans l'ordre où ils interviendront durant l'exposé. Si une lecture linéaire de l'ensemble est sans doute préférable, je m'attacherai à renvoyer le lecteur vers certaines parties informatives lorsque cela sera jugé utile. Notons, également, qu'afin d'éviter d'alourdir l'exposé nous nous autoriserons l'utilisation de certaines abréviations dont l'usage sera spécifié à chaque première occurrence du terme abrégé.

Compte tenu du caractère un peu particulier de ce stage, dans le sens où il a été effectué dans un service de recherche et développement, des libertés circonstanciées ont sciemment été prises sur le plan imposé afin de rendre mieux compte des modes de travail et du déroulement de cette période.

Présentation

Pour faciliter la lecture, les conventions chromatiques suivantes seront utilisées dans ce rapport :

> **Terme :** définition du terme

Note informative

Note d'avertissement

Titre

```
Description technique, il peut s'agir de code Prolog, de scripts,
d'algorithmes ou encore de description d'un point précis du langage Prolog.
Le lecteur peu enclin à lire ce genre de contenu pourra donc les éviter sans
crainte.
```

Prédicat fourni au programmeur par la bibliothèque qui a fait l'objet du stage

1.2 Le choix et le sujet du stage

Le stage de fin d'étude du cycle ingénierie de l'Epita représente une étape décisive dans le cadre de

l'enseignement pédagogique de l'école. Il permet, en effet, de mieux appréhender ce à quoi pourrait ressembler notre vie professionnelle future, avec ce que cela comporte comme joies et désillusions. Il permet aussi d'établir de nouvelles connaissances et de se forger des relations dans notre futur domaine professionnel, ce qui, de toute évidence, est bien loin d'être fortuit dans le monde du travail.

Etant très attiré par la recherche appliquée à l'industrie, je recherchais un stage dans un service orienté recherche et développement. Suite à la conférence de M. Patrick Taillibert portant sur les applications de l'*Intelligence Artificielle* au sein du groupe Thales, j'ai postulé pour différents sujets qui étaient proposés.

Finalement, le sujet du stage, tel qu'il a été proposé et validé dans la convention était le suivant :

« Développement d'une plate-forme multiagent »

Y était associé le commentaire suivant :

« Dans le cadre de ses activités en Intelligence Artificielle, Thales Systèmes Aéroportés a conçu et développé une plateforme de déploiement d'agents intelligents. Cette plateforme facilite considérablement la réalisation d'un système multiagent et son implantation sur différentes machines hôtes ainsi que le déploiement d'agents dont le comportement est décrit en Prolog. Plusieurs applications ont déjà été développées dans les domaines de la société (localisation de cibles, planification de trajectoire, diagnostic automatique…) . L'expérience acquise grâce à cette série d'expérimentations a permis de définir un certain nombre d'évolutions, tant en ce qui concerne les spécifications de la plateforme que son implémentation ; cela nous a conduit à envisager un nouveau développement pour lequel nous proposons le présent stage. Il permettra au stagiaire retenu d'acquérir une bonne expérience du développement logiciel dans un contexte distribué. Le stage couvrira l'ensemble du cycle de vie d'un logiciel, depuis les spécifications jusqu'à la validation finale pour laquelle le stagiaire devra développer un système multiagent de démonstration destiné à mettre en avant les caractéristiques de la plateforme et à aider les nouveaux utilisateurs. »

Bien qu'il n y ait pas eu de modification fondamentale du contenu du stage, nous avons jugé souhaitable de changer le sujet au profit de :

«Réalisation d'une bibliothèque de déploiement des systèmes multiagents »

Ce changement se justifie par l'utilisation inopportune du terme de « plate-forme » qui est plutôt associé à une idée de centralisation. Or, comme nous le verrons, le stage, qui s'est orienté vers une décentralisation maximale de l'information, se situe à un niveau d'abstraction inférieur en ce sens qu'il est possible de développer une plateforme à partir de la bibliothèque qui a fait l'objet du stage. Nous développerons bien sûr davantage ces propos dans une partie ultérieure de ce rapport. Notons simplement, pour l'instant, qu'il s'agissait concrètement de développer une bibliothèque en Prolog dénommée ALBA et permettant la mise en place de systèmes multiagents (SMA).

Le stage a été effectué dans les locaux de Thales à Elancourt[1]. Son déroulement a suivi les grandes étapes classiques de ce genre de stage, qui n'est bien sûr pas sans montrer des similarités avec les grands stades du cycle de vie du logiciel. Si on voulait le résumer de manière succinte et simpliste en quelque mots, cela donnerait à peu près ceci : présentation, prise de contact, mise en place des logiciels de travail, aperçu de la problématique, étude des outils, analyse, spécifications, développement, tests, rédaction de manuels, présentation à l'équipe, assistance et accompagnement des utilisateurs. Notons ici, que du fait de l'orientation recherche du service, ces phases s'articulent avec beaucoup plus de souplesse que dans des structures plus classiques. Toutes ces étapes font bien sûr appel à des compétences diverses et complémentaires que nous serons amenés à développer.

Mais n'anticipons pas! Dans un premier temps, il conviendra de présenter le groupe Thales, d'aborder l'éventail de ses offres, de ses clients, de ses objectifs ce qui nous amènera à le replacer dans son secteur d'activité. Puis nous nous approcherons de plus en plus du service dans lequel le stage a été effectué, ce qui nous conduira à replacer ce dernier dans les travaux de l'entreprise et ainsi à mieux appréhender comment il se positionne au sein de Thales. Une fois le contexte présenté, nous décrirons les conditions humaines et organisationnelles dans lesquelles s'est déroulé le stage. Les aspects scientifiques et techniques seront alors développés ce qui nous donnera l'occasion de détailler le travail accompli, les choix retenus, ainsi que les problèmes et les solutions qui furent adoptés. Fort de ces développements, nous dresserons un bilan de l'expérience. Il s'agira notamment de déterminer, de manière critique et objective, dans quelle mesure le travail réalisé rentre en adéquation avec les attentes initiales de l'entreprise et de dégager qu'elles en furent les apports à titre personnel. Mais rentrons maintenant dans le vif du sujet avec

1 Saint-Quentin-en-Yvelines

une présentation générale qui, à défaut d'être exhaustive, tentera de donner une bonne vue d'ensemble du groupe Thales.

1.3 Présentation de l'entreprise

1.3.1 En bref

Thales, l'un des grands groupes mondiaux de l'électronique, est présent sur 3 marchés : défense, aéronautique et sécurité. Fort d'un chiffre d'affaires de 10,3 milliards d'euros en 2004, Thales emploie 60 000 personnes dans plus de 50 pays. Sa position de leader dans le domaine des hautes technologies est reconnue dans le monde entier.

1.3.2 Historique

1.3.2.1 Les origines

Thales, autrefois connue sous le nom de Thomson-CSF, est issue de la fusion, en 1968, des activités d'électronique professionnelle de Thomson-Brandt avec la Compagnie Générale de Télégraphie Sans Fil (C.S.F.).

Les origines de Thomson-Brandt remontent à 1893, avec la création de la Compagnie Française Thomson- Houston, chargée d'exploiter en France les brevets de la Thomson- Houston Electric Corporation, dans le domaine, alors émergent, de la production et du transport de l'électricité.

Créée en 1918, la C.S.F. a été, avec sa filiale Société Française Radioélectrique (la S.F.R. absorbée en 1957), l'un des pionniers dans le domaine des transmissions hertziennes. Les deux sociétés ont joué un rôle particulièrement important, avant la Seconde Guerre Mondiale, dans le développement de la radiodiffusion, des radiocommunications sur ondes courtes, de l'électro-acoustique et déjà du radar et de la télévision.

1.3.2.2 Les années 70

Dans les années 70, en particulier après les deux chocs pétroliers de 1973 et 1979, Thomson-CSF conclut ses premiers grands contrats à l'exportation avec des pays du Moyen-Orient, notamment pour la vente de ses systèmes de défense anti-aériens Shahine puis Crotale. Cette décennie sera également marquée par la diversification des activités, avec principalement le développement accéléré de la commutation téléphonique, tout d'abord par croissance interne puis, en 1976, avec l'acquisition des filiales françaises de l'Américain ITT (Le Matériel Téléphonique - LMT - et la société Lignes Télégraphiques et Téléphoniques - LTT) et du Suédois Ericsson (Société Française des Téléphones Ericsson). La société participe, également sous l'égide des pouvoirs publics, au développement en France de l'industrie des semi-conducteurs silicium. Et en 1979, elle reprend à sa société mère Thomson-Brandt la société d'imagerie médicale CGR (Compagnie Générale de Radiologie).

1.3.2.3 La nationalisation

A la nationalisation, en février 1982, de Thomson-Brandt, la situation financière de Thomson-CSF est fortement dégradée : à l'issue de la croissance accélérée des années 70, le portefeuille d'activités, très diversifié, inclut de nombreux domaines où la taille et les parts de marché, et donc la rentabilité, de Thomson-CSF sont insuffisantes. Et, malgré les ressources perçues au titres des premiers grands contrats avec des pays du Golfe, l'endettement s'est fortement accru. Les exercices 1982 et 1983 se soldent par de lourdes pertes et à fin 1983, les fonds propres sont négatifs de 337 MF et l'endettement financier net approche les 7 milliards de francs.

Pour assurer le redressement de la situation financière, la nouvelle direction engage rapidement la restructuration du portefeuille d'activités et décide de le focaliser autour du métier cœur, l'électronique de défense. Outre les sorties d'activités périphériques, les désinvestissements majeurs réalisés à partir de 1982 concernent les télécommunications civiles, au travers d'un important accord conclu en 1983 avec la C.G.E., et le médical, cédé à General Electric en 1987 dans le cadre de l'accord organisant également la reprise, par Thomson- S.A., des activités d'électronique grand public de la firme américaine.

Les activités de semiconducteurs, elles aussi très déficitaires au début de la décennie 80 en raison de leur taille "sous-critique", feront, dans un premier temps, l'objet d'un développement interne vigoureux (1), complété en 1985 par l'acquisition, à United Technologies, de sa filiale Mostek. Mais la montée en puissance des concurrents japonais rendant de plus en plus difficile l'accession par croissance interne seule à une part de marché suffisante, Thomson-CSF et le groupe public italien IRI-Finmeccanica décidaient en 1987 de fusionner leurs activités. La joint venture SGS-THOMSON- issue de cette fusion se plaçait dès sa création au deuxième rang des fabricants européens de semi-conducteurs, derrière Philips (voir, plus loin, le paragraphe sur SGS-THOMSON).

Outre l'impact, très favorable sur la rentabilité opérationnelle, de la politique de concentration du portefeuille d'activités, le redressement des résultats a également bénéficié de l'apport très positif des activités financières développées en interne à partir de 1984 pour gérer la trésorerie générée par d'importants contrats conclus à l'exportation. Ces activités financières, regroupées en 1987 au sein de Thomson-CSF Finance, feront l'objet d'un accord conclu à la fin de 1989 avec le Crédit Lyonnais et organisant dès 1990, leur reprise progressive par la Banque, en échange d'une participation de Thomson-CSF au capital de celle-ci. La sortie définitive de Thomson-CSF Finance (rebaptisé Altus) sera effective en juin 1993.

A partir de la fin des années 80 et de la chute du Mur de Berlin, l'environnement des industries de la défense est profondément et durablement altéré. Au déclin des budgets dans la quasi-totalité des pays occidentaux - la France faisant alors figure d'exception - s'ajoute la contraction de la demande dans les pays du Moyen-Orient. La demande solvable émergente, identifiée dans la région du Sud-Est Asiatique, est encore insuffisante pour compenser cette double contraction.

Dès 1987, Thomson-CSF, anticipant la baisse inéluctable des budgets et en prévision de l'achèvement des grands contrats-export alors en cours de facturation, entame la restructuration en profondeur de ses activités : allègement et adaptation des structures, diminution des effectifs, concentration des sites industriels et des moyens de recherche et développement, réorganisation des réseaux commerciaux internationaux, actions pour réduire le coût des approvisionnements,...Les frais de restructuration cumulés sur la période 1987-1995 dépassent 6,5 milliards de francs, un montant qui représente presque 2% du chiffre d'affaires de la période.

Parallèlement, une politique active de croissance externe (acquisitions et alliances au sein de joint ventures) est menée, principalement en Europe, qui permet d'atténuer le déclin des ventes. L'addition des chiffres d'affaires des sociétés entrées dans le périmètre de consolidation depuis 1990 s'élève à 15 milliards de francs (2). L'acquisition la plus importante de la période a été la reprise, finalisée en décembre 1989, des activités d'électronique de défense du groupe Philips, à savoir les sociétés hollandaise Signaal, belge MBLE et française TRT (dénominations sociales modifiées depuis, pour les deux dernières), qui a apporté un chiffre d'affaires de 4,5 milliards de francs en 1990.

Au travers de ces acquisitions, Thomson-CSF a tout d'abord acquis ou consolidé sa présence sur l'ensemble des segments de l'électronique de défense et étoffé son très large spectre, désormais comparable à celui de ses grands concurrents américains. Mais la société a également accru la part de ses activités d'électronique professionnelle civile : sur les 15 milliards acquis, environ 11 milliards concernent l'électronique de défense et 4 milliards les applications civiles.

Enfin, elle a, en quelques années, développé sa présence industrielle hors des frontières nationales : en 1989, 95 % du chiffre d'affaires consolidé provenait des implantations françaises ; six ans plus tard, en 1995, les filiales étrangères ont contribué pour plus de 20 % aux ventes consolidées.

En 1995, pour renforcer la performance de son organisation opérationnelle, Thomson-CSF achève le processus de filialisation initié pour partie quelques années plus tôt.

1.3.2.4 Le processus de privatisation

L'année 1996 marque le début du processus de privatisation. Après avoir annoncé en février 1996 sa volonté de procéder à la privatisation du Groupe, le gouvernement décide en décembre de privatiser Thomson-CSF selon une procédure distincte de celle de Thomson multimédia. En février 1997, il opte pour une privatisation selon une procédure de gré à gré avec cahiers des charges, procédure qui sera abandonnée suite à la dissolution du gouvernement.

C'est finalement en octobre 1997 que le nouveau gouvernement décide de regrouper autour de Thomson-CSF, dans le cadre d'un partenariat stratégique avec Alcatel, les activités d'électronique spatiale et de défense et les activités de communications militaires d'Alcatel, les activités d'électronique professionnelle et de défense de Dassault Electronique, ainsi que les activités satellites d'Aerospatiale.

Le 14 avril 1998 les sociétés Aerospatiale, Alcatel, Dassault Industries, Thomson SA et Thomson-CSF signent l'Accord de Coopération qui définit les modalités de mise en œuvre des orientations décidées par le gouvernement. Enfin, les 22 et 23 juin 1998, l'Assemblée générale des actionnaires et le Conseil d'administration respectivement approuvent les opérations d'apports. Désormais, la majorité du capital de Thomson-CSF, soit 53,06 %, est détenue par des actionnaires du secteur privé.

1.3.2.5 Dualité et Multidomesticité

La dynamique d'implantation "multidomestique" des activités de défense, poursuivie tout au long de la décennie 90, dépasse les frontières du Continent européen : Afrique du Sud, Australie, Corée, Singapour... En juin 2000, l'acquisition par OPA de la société britannique Racal Electronics fait du Royaume-Uni le deuxième pays d'implantation; les activités de défense et les technologies de l'information et services (IT&S) en sont renforcées. Le développement du groupe, par croissance interne et par acquisitions, a aussi profond ément modifié le spectre de ses activités : une réflexion stratégique met en évidence la part croissante des applications civiles et la "dualité" des technologies qui font la force du groupe. En juillet 2000, une nouvelle organisation en trois pôles est mise en place, en cohérence avec la nouvelle approche stratégique. En décembre 2000 Thomson-CSF, devenu Thales, annonce la création avec l'Américain Raytheon de la première joint-venture transatlantique entre industriels de la défense.

2001 marque la poursuite du recentrage des activités : Thales prend le contrôle total de plusieurs de ses joint-ventures en défense et en aéronautique, et se retire d'Alcatel Space; en IT&S, priorié est donnée aux activités synergiques et en forte croissance, principalement le positionnement par satellite et les opérations sécurisées. La structure de l'actionnariat évolue : le flottant frôle les 40%, tandis qu'Alcatel se désengage partiellement et que l'Etat français passe sous la barre des 33%, au profit de l'actionnariat salarié.

1.3.3 Thales de nos jours

1.3.3.1 Thales Group

Le groupe Thales est aujourd'hui composé de 6 divisions, chacune spécialisée dans un domaine précis des compétences du groupe : Aerospace, Air Systems, Land & Joint Systems, Naval, Security et Services.

En 2004, le groupe a consolidé son chiffre d'affaire à 10,3 milliards d'euros et emploi un total de 60.000 personnes dans près de 50 pays. La Recherche & Developpement représente une part importante des activités du groupes avec un budget de 1,85 milliard d'euros en 2004 (soit plus de 17% du chiffre d'affaire) et 18.500 employés dont 70% d'ingénieurs.

1.3.3.2 Thales Aerospace Division

Naissance

Thales Aerospace Division est une nouvelle division qui a vu le jour durant mon stage et qui regroupe deux anciennes divisions : Thales Airborne Systems (au sein de laquelle le stage s'est déroulé) et Thales Avionics afin de rendre ces dernières plus fortes face aux attaques du marché (proposition de rachat de la part de EADS au début de l'année 2005). Ce regroupement s'explique notamment par le fait que les

compagnies aériennes et les forces armées utilisent des systèmes similaires sur leurs avions respectifs. Il n'a donc pas semblé aberrant d'intégrer les expertises systèmes aéroportés et avionique.

Chiffres

Thales Aerospace Division se positionne au premier rang européen et troisième mondial dans son secteur d'activité et compte pour 21% du chiffre d'affaire global du groupe avec près de 2,1 milliards d'euros en 2004 (dont 17% consacré à la Recherche & Développement) avec environ 12.900 employés répartis sur 8 pays (France, Royaume-Uni, Allemagne, Italie, USA, Canada, Chine et Singapour).

Description

La mission principale de la division aéronautique est de fournir à ses clients une gamme complète et innovante d'équipements, de sous-systèmes, de systèmes et de service à forte valeur ajoutée. Les clients du marché civil sont principalement les constructeurs aéronautiques, les compagnies aériennes et les opérateurs. En ce qui concerne le marché militaire, la clientèle est principalement constituée des forces armées, des agences d'armement et des fabricants de plates-formes. Dans le domaine de l'avionique, il s'agit principalement d'apporter toute l'intelligence indispensable à bord de chaque vol dans le ciel et sur terre en se concentrant sur les commandes de vol, la navigation, la communication et la surveillance. Dans le cadre des missions militaires, le but est de mettre en œuvre toutes les capacités critiques indispensables à la maîtrise de l'espace aérien en se focalisant sur les grandes fonctions clés que sont le combat, la surveillance et le renseignement (électronique de cockpit, radars aéroportés, guerre électronique, systèmes de drones, système de mission...). La division aéronautique compte parmi les principaux fournisseurs d'Airbus, de Dassault Aviation et d'Eurocopter. Elle est notamment intervenue dans l'équipement de l'A380, du Boeing 787, du Rafale, du Mirage 2000, du Sea King MK7...

Les objectifs principaux de la division sont principalement d'être le partenaire unique de ses clients, de croître dans la chaîne de valeur (maîtrise d'œuvre, vente d'équipements et de systèmes, services à forte valeur ajoutée) et d'exploiter au mieux la dualité des technologies.

1.3.4 Contexte du stage

L'ensemble de mon stage s'est déroulé dans la division Systèmes Aéroportés du Groupe Thales, plus exactement au sein du service d'étude en *Intelligence Artificielle* de la Direction Technique de la *Business Unit MAS (Mission and Avionic Systems)*.

Ce service s'intéresse à l'application des techniques de l'*Intelligence Artificielle* aux systèmes aéroportés, le but étant de proposer des solutions innovantes et opérationnelles dans les applications de l'entreprise en ce qui concerne l'*intelligence* du traitement de l'information.

Ses axes principaux de recherche sont notamment :

- la **représentation des connaissances**
- le **diagnostic automatique**
- la **planification**
- l'**arithmétique des intervalles**
- la **programmation logique et à contraintes**
- le **raisonnement en temps contraint**
- les **systèmes multiagents**

Les outils développés à l'heure actuelle dans ce contexte sont entre autre:

- Outils de développement de systèmes multiagents
- Logiciel de localisation passive de cibles marines (servant surtout de plate-forme de test des idées en grandeur nature et de démonstrateur des outils réalisés).
- Outil d'aide à la préparation de mission de drones

- Outil de génération automatique de cas de tests de logiciels (permettant de supprimer du code superflu et de traquer les possibles bugs en générant de manière automatisée des cas de test exhaustifs sur l'ensemble des branches des algorithmes programmés).
- Outil permettant d'étudier la conformité des programmes à leur spécification
- Outil de diagnostic automatique de procédés industriels

Il est principalement composé d'étudiants d'écoles d'ingénieurs ou d'universités sous la responsabilité de deux employés de Thales : M. Patrick Taillibert (mon responsable de stage) et M. Bernard Botella ; le nombre important de stagiaires dans le service permet au final à l'entreprise d'avoir une interface optimale avec les nouvelles technologies et les concepts novateurs.

Situé à la frontière entre recherche universitaire et pragmatisme industriel, le service tente de bénéficier au mieux de la collaboration entre université et industrie. Par le biais de ses employés, de ses stagiaires et de ses thésards il est résolument ouvert vers la communauté scientifique du domaine, avec laquelle il intéragit par les moyens suivants :

- **Relations contractuelles avec les laboratoires publics**
- **Comités scientifiques**
- **Jury de thèses**
- **Groupes de travail** : groupe de travail sur les méthodes ensemblistes, groupe automatique-informatique sur le diagnostic, planification, graphes conceptuels, programmation logique
- **Conférences**
- **Commissions de sélection**
- **Comités de programme**
- **Réseaux d'excellence européens** : PLANET (planification), MONET (raisonnement qualitatif et à base de modèles), COMPULOG (programmation logique)
- **Cours**
- **Associations** : American Association for Artificial Intelligence, IEEE computer society, Association Française d'Intelligence Artificielle...

1.4 État des connaissances

1.4.1 Dans l'entreprise

Comme nous l'avons vu le service a acquis une expertise importante dans de nombreux thèmes de l'*Intelligence Artificielle*. M. Patrick Taillibert a conduit ses activités dans ce domaine au sein de Dassault Electronique, de Thomson-CSF Detexis et enfin de Thales depuis plus de 20 ans. Il s'est notamment intéressé aux systèmes multiagents depuis plusieurs années et a contribué au domaine par le biais d'interventions dans le cadre des comités de programme des Journées Francophones sur les Systèmes MultiAgents ou encore d'articles dont notamment :

- 2003, P. Taillibert, *Les systèmes multiagents : une approche pour la coopération entre acteurs intelligents*, COGIS : Commande, Optimisation, Gestion Intelligente, et architecture des Senseurs pour les systèmes Paris (France) 3-4 Juin 2003
- 2004, P. Taillibert, *Multiagent systems for Decision support systems, a point of view*, Workshop Advanced Control and Diagnosis (ACD) Karlsruhe – 18 Nov 2004

Dans ce contexte, différents thèmes ont été ou sont en cours de développement, le plus souvent dans le cadre de thèse CIFRE. Citons ici quelques travaux visant à répondre aux problématiques des applications en développement :

- *Contrôle de systèmes multiagents*
- *Systèmes multiagents et temps réel*
- *Recherche de consensus dans les systèmes multiagents*
- *Débogage des systèmes multiagents*
- *Mémoire d'un agent et graphes conceptuels*

L'utilisation de Prolog a été une constante pour le service depuis de nombreuses années, il est donc évident que dans le domaine de la programmation logique les connaissances et le savoir faire y sont également excellents. L'état pointu d'expertise de l'entreprise, relative au sujet du stage, était donc propice à un encadrement des plus éclairé.

Nous reviendrons sur les raisons qui ont poussées l'équipe à explorer la voie des systèmes multiagents et exposerons plus précisément comment s'articule le sujet dans le service dans une partie ultérieure.

1.4.2 Chez le stagiaire

Dans le cadre de la spécialisation Sciences Cognitives et Informatique Avancé, j'ai eu le loisir de me familiariser avec différentes technologies de l'*Intelligence Artificielle (*IA*)* dont notamment avec les systèmes multiagents qui constituent une tranche non négligeable de la formation. Ils ont d'ailleurs été présentés à l'extrême fin du cursus comme une évolution de ce qui avait été vu antérieurement.

Plusieurs aspects m'intéressaient dans ce sujet. Tout d'abord, il faisait intervenir les systèmes multiagents qui constituent une voie active de recherche dans le domaine de l'IA et qui sont sans doute destinés à connaître un avenir prometteur dans les années à venir.

L'acquisition de compétences dans le développement logiciel en environnement distribué constituait également un argument important en faveur du choix de ce sujet. Il s'agit en effet d'un domaine pointu et en pleine expansion à l'heure où le mot d'ordre général est à la décentralisation des services et des applications à travers des environnements informatiques distribués et hétérogènes. Dans ce contexte, les connaissances acquises à l'école, notamment en programmation réseau, semblaient pouvoir être mises à profit. Nous serons amenés à étudier plus en avant les problèmes que cela pose en pratique et comment l'on a tenté de répondre à cette complexité.

Un autre point d'intérêt résidait dans l'utilisation du langage Prolog pour les développements du stage. N'ayant que de très maigres connaissances en programmation logique et estimant que cette discipline constituait un impératif dans le bagage d'un informaticien spécialisé en IA, le sujet représentait une occasion idéale d'acquérir une bonne maîtrise théorique du langage Prolog et de la mettre en pratique dans le cadre de développements conséquents.

Il était également séduisant de pouvoir suivre l'ensemble du projet et d'intervenir en amont, du stade des spécifications jusqu'à l'aide aux nouveaux utilisateurs. Il apparaissait, en effet, moins contraignant et plus formateur de prendre part à l'ensemble des phases du projet que de poursuivre un développement déjà bien abouti, ce qui est parfois source de difficultés et de problèmes lorsque l'on est obligé de s'adapter à une conception qui ne correspond pas à l'image que l'on s'en fait. Nous aurons toutefois l'occasion de constater qu'à ce propos, l'existant était déjà conséquent, il n'était évidemment pas question de faire table rase du passé mais plutôt de bénéficier de l'expérience déjà acquise par le service dans son domaine afin de tenter de répondre au mieux aux problématiques posées.

Enfin, étant attiré par la recherche appliquée à l'industrie, ce stage présentait une belle opportunité de mieux appréhender le fonctionnement d'un service de recherche et développement au sein d'un grand groupe, même si, comme nous le verrons, le cadre dans lequel s'est effectué le stage est peut-être un peu atypique et pas forcément représentatif de ce qui se fait ailleurs.

1.5 Contexte de travail

Comme nous l'avons déjà mentionné, le stage a été effectué au sein des locaux de Thales à Elancourt. L'environnement de travail était très agréable. Nous étions tous réunis dans une salle très spacieuse facilitant grandement les communications, ce qui s'avère primordial dans le domaine de la recherche où les échanges d'idées et d'informations sont la clé de voûte d'une collaboration productive.

Pour ce qui est des moyens informatiques, nous disposions d'un pentium IV 1.8 GHz sous Windows 2000 professionnel. Les machines étaient équipé de SICStus Prolog 3.11.2 et de Java. Une grande liberté nous a été laissée concernant les logiciels que nous voulions utiliser, dans la mesure où ils étaient libres

évidemment. J'ai, pour ma part, développé sous Emacs et utilisé OpenOffice pour les tâches documentaires. Signalons, de plus, qu'une station Sun Microsystems sous Solaris 8.0 rendait accessible à tous un serveur CVS centralisant l'ensemble des développements de l'équipe, l'ensemble étant sauvegardé quotidiennement en incrémental et intégralement de façon hebdomadaire.

Durant la première partie du stage, nous n'avions pas d'accès direct à Internet, mais, grâce à mon maître de stage cette situation a pu être palliée, ce qui peut s'avérer utile en pratique, même si je n'en ai pas eu un besoin impérieux.

Le travail s'est déroulé dans les meilleurs conditions possibles puisque nous pouvions avoir accès sans difficulté aux documentations internes que nous souhaitions. Il faut, par ailleurs, saluer la disponibilité exemplaire des encadrants du stage qui étaient toujours prêts à répondre aux questions mais également à échanger des idées sur tous les domaines. Outre mon responsable de stage, j'ai également eu la chance de pouvoir échanger beaucoup d'idées avec Mlle Caroline Chopinaud, qui poursuit actuellement sa thèse à Thales, et qui a grandement contribué à l'ancienne version de la bibliothèque de déploiement des agents. Il était ainsi extrêmement appréciable de pouvoir bénéficier de son expérience pratique de manière aussi directe. J'ai, d'autre part, pu échanger des informations par courrier électronique avec certains employés du groupe.

Les effectifs ont beaucoup évolués durant ces six mois. Lorsque j'ai rejoint le service avec deux autres épitéens, François Chazal et Silvère Lhermite, l'équipe se composait donc, en sus de nous, de M. Patrick Taillibert, M. Bernard Botella et Mlle Caroline Chopinaud, auxquels il faut adjoindre par intermittence M. Eric Maes et M. Nicolas Tchitcheck. Nous avons été rejoint en Avril par quatre stagiaires supplémentaires.

Chapitre 2
Aspects organisationnels

2.1 Découpage du projet

Nous avons représenté sur l'organigramme ci-dessous comment le temps du stage a été organisé. Il s'est découpé comme suit :

Apprentissage de Prolog

N'ayant jamais travaillé en Prolog il a été indispensable de se familiariser avec ce langage durant les deux premières semaines du stage.

Spécifications d'ALBA

Environ un mois a ensuite été consacré à l'étude de l'existant. Cette période a permis de comprendre comment les systèmes multiagents étaient utilisés dans le service, de me familiariser avec le code source de l'ancienne version d'ALBA et de reprendre les spécifications de la bibliothèque.

Implémentation du coeur d'ALBA

Une fois le langage Prolog connu et les spécifications d'ALBA validées, le développement à proprement parler des fonctionnalités d'ALBA a duré environ deux mois. Nous nous sommes d'abord concentré sur un coeur orienté monoposte. Une fois, l'implémentation de ce coeur validé, nous nous sommes attelés à la partie multi-machine.

Implémentation de PrologDoc

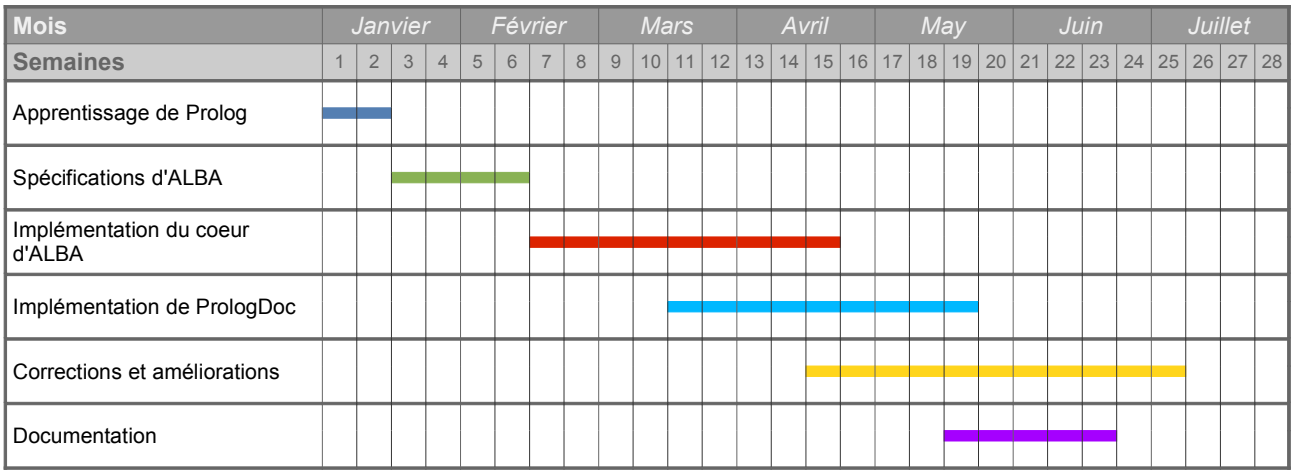
En parallèle du travail sur ALBA, j'ai pris l'initiative de développer PrologDoc qui est un outil de génération de documentation HTML à partir de code source Prolog dûment commenté. Ce travail nous a occupé pratiquement trois mois par intermittence. Il a été mené en équipe.

Corrections et améliorations

Les outils développés étant toujours perfectibles, une période de plus de deux mois a été consacrée aux corrections et améliorations. Il fallait également tenir compte des remarques et critiques des nouveaux stagiaires arrivés en avril.

Documentation

La dernière période du stage a été également consacrée à la rédaction de documentations utilisateurs.



2.2 Respect des délais et critique sur ce découpage

Comme nous l'avons mentionné, le contexte du stage était particulier puisqu'il s'inscrivait dans un service de recherche et développement. Aucun délai n'était réellement imposé par M. Patrick Taillibert et nous étions autonomes quant à l'organisation de notre emploi du temps. L'essentiel était bien sûr que le travail trouve son aboutissement à l'issue du stage.

Avec le recul, les délais ont été bien respectés et le découpage semblait pertinent. En effet, le nécessaire a été fait pour que les stagiaires du mois d'Avril trouvent des outils fonctionnels afin de commencer à travailler dans les meilleurs conditions.

Dans le contexte qui était le nôtre, il eût été illusoire de rédiger des spécifications et de les implémenter en bloc. Nous n'étions pas dans le cadre d'un cahier des charges précis et intangible qui nous aurait été fourni par un client. Il était donc souhaitable de revenir sur les spécifications. De tels développements ne peuvent évidemment jamais être considérés comme complètement achevés car ils sont toujours perfectibles. Le travail demandé à l'origine a toutefois été effectué dans les délais et nous avons même pu travailler sur d'autres outils sans rapport direct avec l'objet du stage.

2.3 Nature et fréquence des points de contrôle

Là encore, les choses se déroulaient de manière assez informelle. Des discussions furent régulièrement menées avec mon maître de stage. Dès qu'un problème de fond se présentait nous faisions une réunion afin d'échanger nos point de vu sur la question et d'envisager de possibles solutions. Nous nous fixions sur un choix technique qui était alors implémenté et présenté pour validation. Les réunions faisaient régulièrement intervenir d'autres stagiaires qui étaient directement concernés par les sujets débattus.

Un point sur l'avancement du travail était également fait environ toutes les deux semaines. Ces points furent plus fréquents aux débuts du développement puis un peu plus espacés dans le temps au fur et à mesure de l'avancée du stage.

Chapitre 3
Aspects Scientifiques et Techniques

Maintenant que nous avons une meilleure vue du contexte dans lequel le stage a été effectué et des conditions organisationnelles qui l'ont régi, il est temps d'en aborder les aspects scientifiques et techniques qui nous retiendrons davantage. Nous tenterons de souligner les problématiques qui se sont posées, d'exposer les choix techniques qui étaient envisageables et de mettre en valeur les solutions retenues et leur justifications. Plutôt que de suivre un plan qui nous aurait conduit à dresser une liste par trop monolithique et cloisonnée, séparant problèmes choix et solutions, il nous a semblé plus judicieux de procéder comme suit.

Nous présenterons, dans un premier temps, un bref état de l'art des systèmes multiagents ainsi que des outils existants pour en faciliter le déploiement. Notons que cet état de l'art est largement tiré de [12] qui nous a semblé cohérent et complet. Le lecteur ayant déjà une bonne connaissance du domaine pourra donc éluder cette partie. Fort de cette étude générale, nous décrirons la vision qu'a le service des sytèmes multiagents, ceci nous amènera à comprendre dans quels buts ils sont utilisés et comment. Il conviendra ensuite de faire une étude de l'existant dont l'analyse nous permettra de mieux cerner le travail qui devait être effectué. Nous aurons ici l'occasion de justifier certains choix techniques imposés pour ce stage (utilisation de SICStus Prolog notamment) et de décrire les outils internes et externes qui ont été utilisés. Nous décrirons enfin le travail qui a été effectué en regroupant dans chaque sous contexte les problèmes rencontrés et les solutions adoptées. Nous nous attacherons, dans la mesure du possible, à partir de considérations générales de définitions et de concepts directeurs puis à poursuivre en diminuant peu à le peu le niveau d'abstraction pour aboutir finalement aux considérations les plus techniques. Cette approche nous permettra de bien suivre tout le déroulement du processus intellectuel et de mieux appréhender comment les idées se concrétisent en pratique.

3.1 Les systèmes multiagents

Le thème des systèmes multiagents est à la connexion de plusieurs domaines en particulier de l'*intelligence artificielle*, des systèmes informatique distribués et du génie logiciel. C'est une discipline qui s'intéresse aux comportements collectifs produits par les interactions de plusieurs entités autonomes et flexibles appelées agents, que ces interactions tournent autour de la coopération, de la concurrence ou de la coexistence entre ces agents.

3.1.1 Des agents autonomes aux systèmes multiagents

3.1.1.1 Qu'est ce qu'un agent

Le concept d'agent a été l'objet d'études dans différentes disciplines. Il a été non seulement utilisé dans les systèmes à base de connaissances, la robotique, le langage naturel et d'autres domaines de l'intelligence artificielle, mais aussi dans des disciplines comme la philosophie et la psychologie. Dans la

littérature, on trouve une multitude de définitions d'agents. Elles se ressemblent toutes, mais diffèrent selon le type d'application pour laquelle est conçu l'agent. De nombreuses discussions ont été menées sur les différentes définitions attribués aux agents ainsi que la différence entre un agent et un programme classique.

À titre d'exemple, voici l'une des premières définitions de l'agent dûe à Ferber :

> **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 multiagent, 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.**

Il ressort de cette définition des propriétés clés comme l'autonomie, l'action, la perception et la communication. D'autres propriétés peuvent être attribuées aux agents. Citons en particulier la réactivité, la rationalité, l'engagement et l'intention.

Récemment, Jennings, Sycara et Wooldridge ont proposé la définition suivante pour un agent :

> **Un agent est un système informatique, situé dans un environnement, et qui agit d'une façon autonome et flexible pour atteindre les objectifs pour lesquels il a été conçu.**
>
> **situé : l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement. Exemples : systèmes de contrôle de processus, systèmes embarqués, etc.**
>
> **autonome : l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne**
>
> **flexible : l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative. Il doit également pouvoir intéragir avec les autres agents (logiciels et humain) quand la situation l'exige afin de compléter ses tâches ou aider d'autres agents à accomplir les leurs.**

Bien entendu, selon les applications certaines propriétés sont plus importantes que d'autres, il peut même s'avérer que pour certaines types d'applications, des propriétés additionnelles soient requises. Il convient cependant de souligner que la présence des propriétés que l'on vient de voir comme l'autonomie, la flexibilité, la sociabilité, etc., donne naissance au paradigme agent tout en le distinguant des systèmes conventionnels comme les systèmes distribués, les systèmes orientés objets et les systèmes experts.

Voyons comment l'idée d'agent a évolué au cours des dernières années.

3.1.1.2 Historique

La majeure partie des composantes d'un agent autonome provient de l'I*ntelligence Artificielle*. On serait donc porté à croire que la construction d'un agent autonome serait au centre des problèmes étudiés par l'IA. Ce n'est cependant pas le cas. En effet, avant les années 80, la recherche en IA s'est plutôt concentrée sur les diverses composantes d'un agent, en supposant qu'on pourrait un jour assembler ces différentes parties afin de construire un agent de manière assez directe. Durant cette période, le champ d'intérêt qui a le plus en commun avec les agents autonomes est celui de la planification.

La planification

La planification est le sous-domaine de l'IA qui cherche à répondre à la question: Que doit-on faire?, c'est-à-dire quelle action poser et dans quel ordre. Comme un agent est une entité qui pose des actions, on voit rapidement l'intérêt de la planification pour les agents autonomes. Un des premiers systèmes de planification à avoir été développé fut le système STRIPS (STanford Research Institute Problem Solver, la distinction entre planificateur et solutionneur n'étant pas encore claire à l'époque). Dans un système STRIPS

typique, on retrouve habituellement les composantes suivantes :

- Un modèle symbolique de l'environnement de l'agent, typiquement dans un sous-ensemble limité de la logique du premier ordre.
- Une spécification symbolique des actions que l'agent peut accomplir, généralement sous forme de pré-condition, action, effet, qui représentent l'état de l'environnement avant que l'action soit posée ainsi que les conséquences de ladite action.
- Un algorithme de planification qui est apte à manipuler les symboles définis et qui génère en sortie, un plan représentant les actions que devra poser l'agent pour atteindre son but.

Des années 70 jusqu'au début des années 80, la recherche sur la planification s'est donc surtout orientée sur les trois aspects exposés ci-dessus. Principalement, on voulait démontrer l'efficacité des algorithmes de planification développés. Malheureusement, on s'est vite aperçu que, bien qu'ils donnent de bons résultats pour des petits problèmes (par exemple, le monde des blocs) les algorithmes de planification ont une performance plutôt médiocre lorsqu'on les applique à des problèmes du monde réel. En effet, comme la taille de l'espace de recherche des algorithmes de planification croit de façon exponentielle avec la complexité de la tâche pour laquelle on cherche à établir un plan, et que les tâches du monde réel sont des tâches très complexes, on obtient par conséquent, un espace de recherche extrêmement large, ce qui donne des temps de recherche proportionnellement longs.

Agents réactifs

Certains chercheurs ont alors commencé à remettre en question le raisonnement symbolique, dont notamment Brooks, qui proposa une alternative que l'on appelle aujourd'hui IA réactive (SMA réactifs). Selon lui, le comportement intelligent devrait émerger de l'interaction entre divers comportements plus simples. Ainsi, au sein de son programme de recherche, il a développé l'architecture subsumption. Dans cette architecture, on bâtit des agents sans utiliser de représentation symbolique ni de raisonnement. Un agent est alors vu comme un ensemble de comportements accomplissant une tâche donnée. Chaque comportement est une machine à états finis qui établit une relation entre une entrée sensorielle et une action en sortie. Le développement d'un agent devient donc un processus où l'on devra expérimenter avec les nouveaux comportements. Ceci est habituellement accompli en plaçant l'agent dans son environnement et en observant les résultats.

Ce type d'approche est généralement utilisé dans le domaine de la simulation. Bien des reproches ont été adressés à cette approche dite « *réactive* », parmi lesquels :

- les agents basent leurs décisions sur des informations locales, il est difficile de voir comment ils pourraient tenir compte des informations non-locales;
- il est difficile de voir comment un agent purement réactif peut apprendre de son expérience et améliorer ainsi ses performances;
- le comportement global d'un agent devrait émerger des interactions entre les divers comportements qui le composent, cette émergence rend donc très difficile la tâche de construire un agent dans le but d'effectuer une tâche spécifique;

Architectures hybrides

Dès le début des années 90, on savait que les systèmes réactifs pouvaient bien convenir pour certains types de problèmes et moins bien pour d'autres. De même, pour la plupart des problèmes, les solutions de l'IA classique, basées uniquement sur la planification, ne conviennent pas non plus. On commence dès lors à investiguer la possibilité de combiner les deux approches afin d'obtenir une architecture hybride. Dans ce cas, un agent est composé de plusieurs couches, arrangées selon une hiérarchie, la plupart des architectures considérant que trois couches suffisent amplement. Ainsi, au plus bas niveau de l'architecture, on retrouve habituellement une couche purement réactive, qui prend ses décisions en se basant sur des données brutes en provenance des senseurs. La couche intermédiaire fait abstraction des donnés brutes et travaille plutôt avec une vision qui se situe au niveau des connaissances de l'environnement. Finalement, la couche supérieure se charge des aspects sociaux de l'environnement, c'est à dire du raisonnement tenant compte des autres agents.

Architectures BDI

Dans le cadre du raisonnement pratique, un raisonnement orienté vers la prise en compte des états mentaux, les chercheurs ont développé l'architecture BDI (de l'anglais *Belief*, *Desire*, *Intention* pour croyance, désir, et intention), une architecture bâtie autour du raisonnement pratique. Ces agents sont généralement représentés par un *état mental* ayant les attitudes mentales suivantes :

- Les croyances : Ce que l'agent connaît de son environnement;
- Les désirs : Les états possibles envers lesquels l'agent peut vouloir s'engager;
- Les intentions : Les états envers lesquels l'agent s'est engagé, et envers lesquels il a engagé des ressources.

Un agent BDI doit donc mettre à jour ses croyances avec les informations qui lui proviennent de son environnement, décider quelles options lui sont offertes, filtrer ces options afin de déterminer de nouvelles intentions et poser ses actions au vu de ses intentions.

3.1.1.3 Les applications des agents autonomes

Les programmes d'aujourd'hui interagissent avec leur usager selon un paradigme appelé manipulation directe. C'est-à-dire que le programme fait exactement ce que l'usager lui demande de faire. Cependant, il pourrait être intéressant d'avoir des programmes qui prennent l'initiative dans certaines circonstances. Ainsi, un logiciel prendrait maintenant une part active pour aider l'usager à accomplir ses tâches. Des applications de tels agents se retrouvent principalement dans les lecteurs de courriels, les lecteurs de groupes de discussions actifs ainsi que les navigateurs actifs.

À part les agents interfaces, la technologie agent se trouve également au niveau des agents mobiles. Un agent mobile est chargé de migrer d'une machine à une autre pour exécuter par exemple un code et ne pas surcharger le réseau.

En fait, l'agent en tant qu'entité individuelle peut s'avérer limité dans bien des cas, surtout au vu de ce qu'on voit aujourd'hui comme applications distribuées et pour lesquelles un ensemble d'agents mettant en commun compétences et connaissances paraît plus que nécessaire. Ce type d'agents forment ce que l'on appelle des systèmes multiagents.

3.1.1.4 Les systèmes multiagents

Un système multiagent est un système distribué composé d'un ensemble d'agents. Les SMA sont conçus et implantés idéalement comme un ensemble d'agents interagissants, le plus souvent, selon des modes de coopération, de concurrence ou de coexistence.

Un SMA est généralement caractérisé par :

- chaque agent a des informations ou des capacités de résolution de problèmes limitées, ainsi chaque agent a un point de vue partiel;
- il n'y a aucun contrôle global du système multiagent;
- les donnés sont décentralisées;
- le calcul est asynchrone.

Les SMA sont des systèmes idéaux pour représenter des problèmes possédant de multiples méthodes de résolution, de multiples perspectives et/ou de multiples résolveurs. Ces systèmes possèdent les avantages traditionnels de la résolution distribuée et concurrente de problèmes comme la modularité, la vitesse (avec le parallélisme), et la fiabilité (dûe à la redondance). Ils héritent aussi des bénéfices envisageables de l'IA comme le traitement symbolique (au niveau des connaissances), la facilité de maintenance, la réutilisation et la portabilité mais surtout, ils ont l'avantage de faire intervenir des schémas d'interaction sophistiqués. Les types courants d'interaction incluent la coopération (travailler ensemble à la résolution d'un but commun) ; la coordination (organiser la résolution d'un problème de telle sorte que les interactions nuisibles soient évitées et que les interactions bénéfiques soient exploitées) ; et la négociation (parvenir à un accord acceptable pour toutes les parties concernées).

Bien que les SMA offrent de nombreuses potentialités, ils soulèvent de nombreux défis :

- formulation, allocation de problèmes, synthèse des résultats
- cohérence des actions

- coordination entre les agents
- résolution des conflits
- problèmes de temporalité
- ...

Les SMA sont à l'intersection de plusieurs domaines scientifiques : informatique répartie, génie logiciel, intelligence artificielle, vie artificielle. Ils s'inspirent également d'études issues d'autres disciplines connexes notamment la sociologie, la psychologie sociale, les sciences cognitives et bien d'autres. C'est ainsi qu'on les trouve parfois à la base des :

- systèmes distribués
- interface homme-machine
- bases de données et bases de connaissances distribuées coopératives
- systèmes pour la compréhension du langage naturel
- protocoles de communication, réseaux
- programmation orientée agents, génie logiciel
- robotique cognitive
- applications distribués, contrôle de trafic routier, contrôle aérien, réseaux d'énergie...

Les recherches dans le domaine des systèmes multiagents poursuivent deux objectifs majeurs : le premier concerne l'analyse théorique et expérimentale des mécanismes qui ont lieu lorsque plusieurs entités autonomes interagissent, le second s'intéresse à la réalisation de programmes distribués capables d'accomplir des tâches complexes via la coopération et l'interaction. Leur position est donc double : d'un côté elles se placent au sein des sciences cognitives, des sciences sociales et des sciences naturelles pour à la fois modéliser, expliquer et simuler des phénomènes naturels, et susciter des modèles d'auto-organisation; de l'autre côté, elles se présentent comme une pratique, une technique qui vise la réalisation de systèmes informatiques complexes à partir des concepts d'agent, de communication, de coopération et de coordination d'actions.

Les agents autonomes et les systèmes multiagents représentent une nouvelle approche pour l'analyse, la conception et l'implantation des systèmes informatiques complexes. La vision basée sur l'entité agent offre un puissant répertoire d'outils, de techniques, et de métaphores qui y ont le potentiel d'améliorer considérablement les systèmes logiciels.

La technologie agent va, sans doute, représenter durant les prochaines années, un nouveau paradigme de programmation pour le génie logiciel. Un nouveau paradigme *similaire* à la programmation orientée objet et qui d'ailleurs pourrait s'intituler programmation orientée agent.

À ce propos, il convient de ne pas confondre *agent* et *objet*. Tout d'abord, tout comme les agents, les objets encapsulent leur état interne (leurs données). Ils peuvent également poser des actions sur cet état par le biais de leurs méthodes et ils communiquent en s'envoyant des messages. À ce niveau, ils diffèrent des agents par leur degré d'autonomie. En effet, une méthode doit être invoquée par un autre objet pour pouvoir accomplir ses effets. Un agent, quant à lui, recevra une requête et décidera de son propre gré s'il doit poser ou non une action. Une seconde différence provient du caractère flexible (réactif, pro-actif et social) du comportement d'un agent. Bien que certains diront qu'il est possible de bâtir un programme orienté-objet qui intègre ces caractéristique, on doit également voir que le modèle standard d'un objet ne dit rien à propos de ces types de comportements. La troisième et dernière différence provient du fait qu'on considère un agent comme étant lui-même une source de contrôle au sein du système tandis que dans un système orienté-objet, on n'a qu'une seule source de contrôle.

Nous dressons maintenant un bref historique du domaine relatif aux SMA, en présentant des travaux précurseurs à ces systèmes ainsi que quelques exemples des premières applications développées.

Historique

En 1980, un groupe de chercheurs s'est réuni pour discuter des défis concernant la résolution *intelligente* de problèmes dans un système comportant plusieurs résolveurs de problèmes. Lors de cette réunion, il a été décidé que l'IA distribuée n'axerait pas ses travaux sur les détails de bas-niveau de la parallélisation ni sur comment paralléliser les algorithmes centralisés mais plutôt sur le fait de savoir comment un groupe de solutionneurs de problèmes pourrait coordonner ses efforts afin de résoudre des

problèmes de manière efficace.

On peut dire que les SMA ont vu le jour avec l'avènement de l'« *Intelligence Artificielle Distribuée* » (IAD). À ses début toutefois, l'IAD ne s'intéressait qu'à la coopération entre résolveurs de problèmes afin de contribuer à résoudre un but commun. Pour y parvenir, on divisait en général, un problème en sous problèmes, et on allouait ces sous-problèmes à différents résolveurs appelés à coopérer pour élaborer des solutions partielles. Celles-ci sont finalement synthétisées en une réponse globale au problème de départ. Ainsi donc, l'IAD au départ privilégiait le problème à résoudre tout en mettant l'accent sur la résolution d'un tel problème par de multiple entités intelligentes. Dans les SMA d'aujourd'hui, les agents sont (entre autres) autonomes, possiblement préexistants et généralement hétérogènes. Dans ce cas, l'accent est plutôt mis sur le fait de savoir comment les agents vont coordonner leurs connaissances, buts et plans pour agir et résoudre des problèmes.

Parmi les systèmes développés tout au début, il y avait :

- Le modèle des acteurs : Un des premiers modèles proposé à l'époque de l'IAD fut le modèle des Acteurs. Les acteurs sont des composantes autonomes d'un système qui communiquent par messages asynchrones. Ils sont composés d'un ensemble de primitives. Les acteurs s'avèrent être un modèle assez naturel pour le calcul parallèle. Cependant, les divers modèles d'acteurs, comme bien d'autres modèles de l'IAD font face à un problème de cohérence. Leur granularité fine pose des problèmes de comportement dans des systèmes qui renferment plusieurs acteurs. Ils éprouvent également des difficultés à atteindre des buts globaux avec seulement des connaissances locales.
- Le protocole Contract Net : Le protocole Contract Net fut une des premières solutions au problème d'allocation de tâches auquel fait face généralement un ensemble de résolveurs de problèmes. Dans ce protocole, les agents peuvent prendre deux rôles: gestionnaire ou contracteur. L'agent qui doit exécuter une tâche donnée (le gestionnaire) commence tout d'abord par décomposer cette tâche en plusieurs sous-tâches. Il doit ensuite annoncer les différentes sous-tâches au reste des agents de l'environnement. Les agents qui reçoivent une annonce de tâche à accomplir peuvent ensuite faire une proposition devant refléter leur capacité à remplir cette tâche. Le gestionnaire rassemble ensuite toutes les propositions qu'il a reçues et alloue la tâche à l'agent ayant fait la meilleure proposition.

Les premières applications

Parmi les premières applications développées à l'aide de la technologie multiagents, on retrouve une application dans le contrôle du trafic aérien et une autre dans la surveillance de véhicules motorisés. Pour le contrôle de trafic aérien, Cammarata a étudié des stratégies de coopération pour pouvoir résoudre les conflits entre les plans d'un groupe d'agents. Ces stratégies ont pu être ensuite appliquées dans le cadre d'un système de contrôle du trafic aérien. Dans ce système, chaque agent (représentant un avion) cherche à bâtir un plan de vol qui devrait le garder à une distance sécuritaire de chaque autre agent. Dans le cas où des agents se retrouveraient dans une situation conflictuelle, ils doivent alors choisir parmi eux, un agent apte à élaborer un nouveau plan de vol sans engendrer de nouveaux conflits. Pour faire ce choix, Cammarata et ses collègues ont opté pour deux stratégies qu'ils ont comparées entre elles. La première a consisté à choisir l'agent le mieux informé ou l'agent le moins contraint pour jouer le rôle de planificateur central et élaborer un nouveau plan de vol qui résoudrait le conflit. La deuxième a consisté à faire le partage suivant: il revient à l'agent le mieux informé d'élaborer un nouveau plan de vol et à l'agent le moins contraint de l'exécuter.

La surveillance de véhicules motorisés tournait, quant à elle, autour du système DVMT (Distributed Vehicle Monitoring Task). Ce système avait comme tâche principale d'identifier quel type de véhicule circulait dans la zone où étaient placés ses senseurs. À partir de ces interprétations, il devait tenter d'établir une description des mouvements des véhicules dans la région qu'il supervisait. Comme bien d'autres systèmes multiagents de l'époque, le système DVMT utilisait un blackboard pour coordonner les efforts de ses différents agents. Rappelons qu'un blackboard est simplement une structure de données partagées entre divers agents. Ces derniers peuvent la consulter pour obtenir des informations sur l'état actuel du problème ou y écrire la partie de la solution qu'ils ont obtenue.

3.1.2 Intéractions et coopération entre agents

Un système multiagent (SMA) se distingue d'une collection d'agents indépendants par le fait que les agents interagissent en vue de réaliser conjointement une tâche ou d'atteindre conjointement un but particulier. Les agents peuvent interagir en communiquant directement entre eux ou par l'intermédiaire d'un autre agent ou même en agissant sur leur environnement. Chaque agent peut être caractérisé par trois dimensions : ses buts, ses capacités à réaliser certaines tâches et les ressources dont il dispose. Les interactions des agents d'un SMA sont motivées par l'interdépendance des agents selon ces trois dimensions: leurs buts peuvent être compatibles ou non; les agents peuvent désirer des ressources que les autres possèdent; un agent X peut disposer d'une capacité nécessaire à un agent Y pour l'accomplissement d'un des plans d'action de Y.

Dans le cadre des systèmes multiagents on doit traiter la question fondamentale qui est l'implantation de la coopération entre les agents. Il a été proposé quatres buts génériques pour établir la coopération dans un groupe d'agents : augmenter le taux de finalisation des tâches grâce au parallélisme, augmenter le nombre de tâches réalisables grâce au partage de ressources (information, expertise, dispositifs physiques, etc.), augmenter les chances de finaliser des tâches en les dupliquant et en utilisant éventuellement des modes de réalisation différents; diminuer les interférences entre tâches en évitant les interactions négatives.

En fait, on peut caractériser un système par le type de coopération mis en oeuvre qui peut aller de la coopération totale à l'antagonisme total. Des agents totalement coopératifs peuvent changer leurs buts pour répondre aux besoins des autres agents afin d'assurer une meilleure coordination entre eux. Cela peut résulter en des coûts de communication élevés.

Les agents antagonistes par contre, ne vont pas coopérer et dans ce cas, leurs buts respectifs vont se trouver bloqués. Dans de tels systèmes, les coûts de communication sont minimaux. La plupart des systèmes réels se situe entre les deux extrêmes : coopération totale et antagonisme total. La coopération totale est mise en oeuvre par exemple dans les approches de résolution coopérative de problèmes distribués. Dans ces approches, les agents coopèrent pour résoudre des problèmes qu'ils ne peuvent pas résoudre individuellement. Tout d'abord chaque agent utilise ses connaissances et ressources pour résoudre localement un ou plusieurs sous-problèmes. Les solutions partielles à tous les sous-problèmes sont par la suite intégrées. C'est par exemple le cas des avions qui volent dans un même espace aérien et qui, de ce fait, ont différentes perceptions de leur espace environnant obtenues par leurs senseurs respectifs. Ce n'est qu'en combinant ces différentes perceptions que ces avions peuvent obtenir une vue globale afin de résoudre les conflits potentiels qui peuvent se produire entre eux.

Ferber remarque que les chercheurs ont développé différents points de vue sur la coopération. Ainsi, on peut considérer la coopération comme une attitude adoptée par les agents qui décident de travailler ensemble ou on peut adopter le point de vue d'un observateur extérieur au système multiagent qui interprète a posteriori les comportements des agents pour les qualifier de coopératifs ou non suivant des critères préétablis tels que l'interdépendance des actions ou le nombre de communications effectuées. Dans le cas de la coopération vue comme une attitude intentionnelle, les agents s'engagent dans une action après avoir identifié et adopté un but commun. C'est par exemple le cas d'un groupe de personnes acceptant de travailler dans une entreprise.

3.1.2.1 Intéractions entre agents coopératifs

Au fur et à mesure que croît l'intérêt pour les applications composées d'agents s'entraidant pour l'atteinte d'un but commun, et que de plus en plus d'agents sont construits dans le but de coopérer avec d'autres agents, il devient important de bien comprendre les principes qui soustendent la coopération.

La planification pour un seul agent, telle que précédemment décrite, ne fait que construire une séquence d'actions en ne considérant que les buts de l'agent, ses capacités et les contraintes imposées par son environnement. Par contre, dans un environnement multiagent, on se doit de tenir compte des contraintes que les actions des autres agents placent sur le choix des actions de l'agent, des contraintes que les engagements de l'agent envers les autres imposent sur son propre choix d'action, ainsi que l'évolution imprévisible de l'environnement due à des agents qui ne sont pas modélisés.

Les premiers travaux en *intelligence artificielle* distribuée étaient principalement concentrés sur des groupes d'agents qui poursuivaient des buts communs. On choisit alors de privilégier une approche selon laquelle on planifie avant d'agir. Ainsi, afin de produire un plan cohérent, les agents doivent être en mesure de reconnaître les interactions entre les différents sous-buts pour pouvoir, soit les ignorer, soit les résoudre.

Une autre approche intéressante pour résoudre les interdépendances entre les sous-buts a été le *Functionnally Accurate Model* dans lequel les agents n'ont pas besoin d'avoir toutes les informations nécessaires pour résoudre les conflits localement. Les interactions se font plutôt par un échange asynchrone de résultats partiels. À partir de ce modèle sont apparus plusieurs principes de coordination.

La Planification Partielle Globale est une approche flexible qui permet aux divers agents d'un système de se coordonner dynamiquement. Les agents interagissent en se communiquant leurs plans et leurs buts selon un niveau d'abstraction approprié. Ces communications permettent à chacun d'anticiper quelles seront les actions futures d'un ou de plusieurs autres agents, augmentant ainsi la cohérence de l'ensemble des agents. Comme les agents coopèrent, le receveur d'un message peut utiliser les informations reçues afin d'ajuster sa propre planification.

Un autre courant de recherche sur la coordination des agents consiste à modéliser explicitement le travail d'équipe qu'effectuent les agents. Cette approche s'avère très pratique dans des environnements dynamiques où les membres de l'équipe peuvent échouer dans leur tâches ou découvrir de nouvelles opportunités. Dans de telles situations, l'équipe doit pouvoir évaluer sa performance et être en mesure de se réorganiser en conséquence. Dans cet ordre d'idées, on parle alors du modèle des *intentions conjointes*. Un tel modèle est d'ailleurs une extension naturelle du raisonnement pratique. Dans ce modèle, on cherche à modéliser l'état mental d'une équipe en action. L'équipe aura une intention conjointe si chaque membre de l'équipe est engagé à compléter l'action de l'équipe. Préalablement à cette intention collective, tous les membres de l'équipe auront établi les croyances et les engagements communs appropriés, et ce en s'échangeant des actes (du langage naturel) de requête et de confirmation.

3.1.2.2 Intéractions entre agents égo-centrés

Les interactions entre agents égo-centrés se basent principalement sur la négociation. Cette dernière devient donc une méthode de coordination et de résolution de conflits. Elle a également été utilisée comme métaphore pour la communication de changements dans les plans, l'allocation de tâches ainsi que la résolution centralisée de violation de contraintes. On peut donc voir que la définition de la négociation est aussi imprécise que celle du concept d'agent. On arrive malgré tout à retrouver certaines caractéristiques importantes de la négociation : la présence d'une forme de conflit qui doit être résolu de façon décentralisée par des agents égo-centrés ayant une rationalité limitée et des informations incomplètes.

Le système PERSUADER de Sycara et les travaux de Rosenschein sont les premiers travaux de l'IAD sur la négociation entre agents égo-centrés. Le système PERSUADER puise son modèle dans les négociations de conventions collectives. Les négociations impliquent donc trois agents (un syndicat, une compagnie et un médiateur) et s'inspirent des négociations entre êtres humains. On y modélise les multiples itérations pendant lesquelles les parties impliquées s'échangent offres et contreoffres, et l'utilité des agents est multidimensionnelle et privée. Les croyances des agents sont révisées de telle sorte qu'on peut atteindre une entente grâce à une argumentation persuasive.

Rosenschein, quant à lui, base son travail sur la théorie des jeux. L'utilité est le seul paramètre considéré par les agents, et on suppose que les agents sont omniscients. Les valeurs de l'utilité des différentes alternatives sont représentées par une matrice des gains qui est commune aux parties impliquées dans la négociation. Chaque partie choisira donc l'alternative qui maximisera son utilité. Malgré l'élégance mathématique de la théorie des jeux, les négociations telles que modélisées par Rosenschein s'appliquent mal à des problèmes du monde réel.

Krauss de son côté s'est penché sur le rôle du temps dans les négociations. Cette étude lui a permis de voir que l'importance qu'un agent accorde au temps peut affecter l'entente finale entre les parties.

3.1.2.3 Coordination entre agents

De nombreux exemples de coordination existent dans la vie quotidienne : deux déménageurs déplaçant un meuble lourd, deux jongleurs échangeant des balles avec lesquelles ils jonglent, des personnes qui parlent à tour de rôle en se passant un micro, etc. Malone note que deux des composantes fondamentales de la coordination entre agents sont l'allocation de ressources rares et la communication de résultats intermédiaires. Dans ce contexte, les agents doivent être capables de communiquer entre eux de façon à pouvoir échanger les résultats intermédiaires. Pour l'allocation des ressources partagées, les agents

doivent être capables de faire des transferts de ressources. Ceci peut d'ailleurs imposer certains comportements à des agents particuliers. Malone conclut qu'il peut être utile de distinguer les liens de contrôle comme une catégorie spéciale de liens de communication par lesquels certains agents transmettent des instructions que d'autres vont être motivés à suivre.

En étudiant les communautés humaines, Mintzberg a identifié trois processus fondamentaux de coordination : ajustement mutuel, supervision directe et coordination par standardisation. L'ajustement mutuel est la forme de coordination la plus simple qui se produit quand deux ou plusieurs agents s'accordent pour partager des ressources en vue d'atteindre un but commun. Habituellement les agents doivent échanger de nombreuses informations et faire plusieurs ajustements à leurs propres comportements en tenant compte des comportements des autres agents. Dans cette forme de coordination aucun agent n'a un contrôle sur les autres agents et le processus de décision est conjoint. La coordination dans les groupes de pairs (*peer groups*) et dans les marchés (*market*) est habituellement une forme d'ajustement mutuel. La supervision directe apparaît quand un ou plusieurs agents ont déjà établi une relation dans laquelle un des agents a un contrôle sur les autres. Cette relation est habituellement établie par ajustement mutuel comme par exemple dans le cas d'un employé ou d'un sous-contractant qui accepte de suivre les instructions du superviseur. Dans cette forme de coordination l'agent superviseur contrôle l'utilisation des ressources partagées (comme par exemple les ressources humaines, le temps de calcul ou l'argent) par les agents subordonnés. Il peut aussi imposer certains comportements. Dans les cas de coordination par standardisation le superviseur coordonne les activités en établissant des procédures que doivent suivre les subordonnés dans des situations identifiées. On trouve par exemple de telles procédures dans les entreprises, mais aussi dans les systèmes informatiques.

Malone suggère qu'en utilisant ces processus de coordination fondamentaux, il est possible de construire des systèmes de coordination sophistiqués dont les plus répandus sont les hiérarchies et les marchés basés respectivement sur les processus de supervision directe et d'ajustement mutuel. Le travail en petit groupe se prête bien à l'ajustement mutuel. Par contre, dès que la taille du groupe grandit et que le nombre de tâches augmente, le nombre de liens et la quantité d'informations échangées peuvent devenir rapidement un handicap sérieux. Un groupe important peut être partagé efficacement en sous-groupes de manière à ce que la plupart des interactions s'effectuent dans les sous-groupes et que les quelques interactions nécessaires entre les sous-groupes soient prises en charge par les superviseurs de ces sous-groupes. Les sous-groupes peuvent être coordonnés par contrôle hiérarchique ou par ajustement mutuel suivant les caractéristiques du domaine d'application et des tâches à accomplir. Les marchés sont aussi considérés comme une activité d'organisation de groupe basée sur l'ajustement mutuel dans laquelle chaque agent contrôle des ressources rares (par exemple la main d'oeuvre, les matières premières, les marchandises et l'argent). Les agents s'accordent pour partager leurs ressources respectives afin d'atteindre un but commun. Les ressources échangées ont un prix explicite ou implicite. Lorsqu'un contrat est conclu, il y a un accord pour que l'agent contracteur devienne le superviseur de l'agent contractant. De nombreuses organisations mettent en oeuvre des processus de coordination mixtes basées sur l'ajustement mutuel, la supervision directe et la standardisation. Ces cadres organisationnels peuvent être des sources d'inspiration quand on doit établir une structure organisationnelle pour un SMA.

Bien entendu, la coordination est une question centrale pour les SMA et la résolution de systèmes distribués. En effet, sans coordination un groupe d'agents peut dégénérer rapidement en une collection chaotique d'individus. On pourrait penser que la façon la plus simple de s'assurer un comportement cohérent du groupe d'agents serait de le faire par un agent centralisateur qui détiendrait des informations de haut niveau sur ces agents. Ainsi, l'agent centralisateur pourrait créer des plans d'action et assigner les tâches aux divers agents du groupe. Cette approche est pratiquement impossible à mettre en oeuvre dans des applications réalistes en raison de la difficulté de réaliser un tel agent centralisateur qui puisse tenir compte des buts, des connaissances et des activités de chaque agent : la charge en communication serait énorme, sans compter qu'on perdrait les avantages d'un SMA composé d'agents autonomes. Le contrôle et les informations doivent alors être distribués parmi les agents.

À cette fin, Jennings propose un modèle qui se base sur l'hypothèse de la centralité des engagements et des conventions. Les engagements sont vus comme des promesses en vue de réaliser certaines actions, alors que les conventions constituent le moyen de faire le suivi de ces engagements dans des circonstances changeantes. Grâce aux engagements des autres agents, un agent peut prédire les actions qu'il est susceptible d'effectuer et déterminer ainsi les interdépendances avec ses propres actions. Mais comme le monde extérieur et les croyances des agents évoluent constamment, un agent doit aussi posséder un moyen pour déterminer si les engagements existants sont encore valides. Les conventions offrent un tel mécanisme en définissant les conditions sous lesquelles les engagements doivent être

réévalués et en spécifiant les actions à entreprendre dans de telles circonstances.

Du point de vue du concepteur d'un SMA diverses questions doivent être traitées comme notamment : Avec quels agents un agent doit-il coordonner ses actions? Quand et où ces actions de coordination doivent-elles être accomplies? Comment détecter et traiter les interactions entre actions (conflits et renforcement)? Détecter les relations existant entre des actions est une activité nécessaire lorsqu'on veut coordonner ces actions.

Von Martial de son côté, a identifié deux grandes catégories de relations pouvant exister entre les actions accomplies simultanément par plusieurs agents : les relations négatives et les relations positives. Les relations négatives (ou conflictuelles) sont celles qui gênent ou empêchent plusieurs actions de s'accomplir simultanément et sont dues en général à des incompatibilités de buts ou des conflits de ressources. Par exemple dans une vente aux enchères, agent X et agent Y veulent acquérir un même meuble M . Les relations positives (ou synergiques) sont celles qui permettent aux actions de bénéficier les unes des autres. Ainsi, la réalisation d'une action a accomplie par l'agent Y réalise du même coup une action x = b que devait accomplir l'agent X ou favorise la réalisation d'une action c par l'agent Z . Par exemple X , Y et Z sont dans une pièce dont les fenêtres sont fermées et les stores baissés. X a chaud et Z aimerait avoir de la lumière. Y monte les stores et ouvre une fenêtre.

3.1.2.4 Négociation entre agents

La négociation joue un rôle fondamental dans les activités de coopération en permettant aux personnes de résoudre des conflits qui pourraient mettre en péril des comportements coopératifs. Durfee et ses collègues définissent la négociation comme le processus d'améliorer les accords (en réduisant les inconsistances et l'incertitude) sur des points de vue communs ou des plans d'action grâce à l'échange structuré d'informations pertinentes. En général les chercheurs en IA distribuée utilisent la négociation comme un mécanisme pour coordonner un groupe d'agents. Différentes approches ont été développées en s'appuyant sur la riche diversité des négociations humaines dans divers contextes.

Un des protocoles les plus étudiés pour la négociation s'appuie sur une métaphore organisationnelle. Le protocole du réseau contractuel (*Contract-Net*) a été une des approches les plus utilisées pour les SMA. Les agents coordonnent leurs activités grâce à l'établissement de contrats pour atteindre des buts spécifiques. Un agent, agissant comme un gestionnaire décompose son contrat (une tâche ou un problème) en sous-contrats qui pourront être traités par des agents contractants potentiels. Le gestionnaire annonce chaque sous-contrat sur un réseau d'agents. Les agents reçoivent et évaluent l'annonce. Les agents qui ont les ressources appropriées, l'expertise ou l'information requise envoient au gestionnaire des soumissions qui indiquent leurs capacités à réaliser la tâche annoncée. Le gestionnaire évalue les soumissions et accorde les tâches aux agents les mieux appropriés. Ces agents sont appelés des contractants. Enfin, gestionnaires et contractants échangent les informations nécessaires durant l'accomplissement des tâches.

Un autre important protocole de négociation a été proposé par Cammarata et ses collègues qui ont étudié les stratégies de coopération pour résoudre des conflits entre des plans d'un ensemble d'agents. Ces stratégies ont été appliquées au domaine du contrôle de trafic aérien avec le but de permettre à chaque agent (représentant un avion) de construire un plan de vol qui permettrait de garder une distance sécuritaire par rapport aux autres avions et de satisfaire des contraintes telles qu'atteindre la destination désirée avec une consommation de carburant minimale. La stratégie choisie, appelée *centralisation de tâche* permettait aux agents impliqués dans une situation conflictuelle potentielle (des avions se rapprochant trop compte tenu de leurs caps respectifs) de choisir l'un d'eux pour résoudre le conflit. Cet agent agissait comme un planificateur centralisé et développait un plan multiagent qui spécifiait les actions concurrentes de tous les avions impliqués. Les agents utilisaient la négociation pour déterminer qui était le plus apte à réaliser le plan. Cette aptitude était évaluée à partir de divers critères permettant d'identifier par exemple l'agent le mieux informé ou celui qui était le plus contraint. Les protocoles de négociation précédents supposent que les agents sont coopératifs, et donc qu'ils poursuivent un but commun.

3.1.2.5 Communications entre agents

Les agents peuvent interagir soit en accomplissant des actions linguistiques (en communiquant entre eux), soit en accomplissant des actions non-linguistiques qui modifient leur environnement. En communiquant, les agents peuvent échanger des informations et coordonner leurs activités. Dans les SMA

deux stratégies principales ont été utilisées pour supporter la communication entre agents: les agents peuvent échanger des messages directement ou ils peuvent accéder à une base de données partagées (appelée tableau noir ou *blackboard*) dans laquelle les informations sont postées. Les communications sont à la base des interactions et de l'organisation sociale d'un SMA.

Dans certains cas, un agent peut inférer les plans des autres agents sans avoir à communiquer avec eux. Dans Tubbs ce mécanisme est appelé *marchandage tacite* et il semble fonctionner au mieux quand les buts des agents ne sont pas conflictuels. Pour étudier ce mécanisme, Rosenschein et ses collègues ont utilisé une approche de la théorie des jeux caractérisée par des matrices de gains qui contiennent les gains des agents pour chacune des issues possibles de l'interaction. Cette approche suppose que les agents partagent la connaissance de cette matrice de gains, ce qui n'est pas réaliste si les agents ne sont pas bienveillants.Rosenschein a étendu cette approche afin de permette aux agent de raffiner leurs choix en tenant compte de l'incertitude des choix des autres agents. Cette approche présente aussi des difficultés car le fait que les agents aient à spéculer sur les choix des autres agents conduit à développer des mécanismes de raisonnement complexes qui peuvent nécessiter une quantité de calculs importante. Il est clair que l'absence de communication doit être compensé par des mécanismes de raisonnement plus complexes. Certains chercheurs ont examiné l'utilisation de communications primitives restreintes à l'utilisation d'un ensemble fini de signaux (souvent au nombre de deux) avec des interprétations fixes « à la Hoare ». Cette stratégie a été employée par Georgeff pour éviter des conflits dans une planification multiagent. Cependant, des informations plus sophistiquées comme des requêtes ou des commandes ne peuvent pas être exprimées avec de tels signaux.

Echange d'informations grâce à un tableau noir

En *intelligence artificielle* la technique du tableau noir est très utilisée pour spécifier une mémoire partagée par divers systèmes. Dans un SMA utilisant un tableau noir, les agents peuvent écrire des messages, insérer des résultats partiels de leurs calculs et obtenir de l'information. Le tableau noir est en général partitionné en plusieurs niveaux qui sont spécifiques à l'application. Les agents qui travaillent sur un niveau particulier peuvent accéder aux informations contenues dans le niveau correspondant du tableau noir ainsi que dans des niveaux adjacents. Ainsi, les données peuvent être synthétisées à n'importe quel niveau et transférées aux niveaux supérieurs alors que les buts de haut niveau peuvent être filtrés et passés aux niveaux inférieurs pour diriger les agents qui oeuvrent à ces niveaux. Le transfert de messages et les communications basées sur un tableau noir sont souvent combinés dans des systèmes complexes. Dans de tels systèmes, chaque agent est composé de plusieurs sous-systèmes qui communiquent à travers un tableau noir. Les agents communiquent entre eux par échange de messages.

Actes de discours et conversations

Dès lors que l'on s'intéresse au contenu des messages échangés par les agents, il est pertinent d'examiner les nombreux travaux qui ont été réalisés sur la compréhension du langage naturel et en particulier les recherches sur l'intentionnalité dans les communications. Dans ces travaux on considère que les agents sont dotés de structures de données appelées *états mentaux* à partir desquelles ils peuvent raisonner. Diverses catégories d'états mentaux ont été retenues par les chercheurs dont notamment les croyances, les désirs, les intentions ou buts, les capacités, etc. Ce sont ces mêmes états mentaux qui caractérisent les modèles des agents cognitifs en *intelligence artificielle* distribuée. Ainsi, un agent cognitif raisonne sur ses croyances qui représentent sa compréhension du monde dans lequel il évolue. Il raisonne aussi sur ses désirs et intentions en relation avec ses croyances et capacités afin de prendre des décisions auxquelles sont associés les plans qu'il va accomplir pour agir dans le monde. Lorsqu'un agent X communique avec un autre agent Y , c'est pour influencer les états mentaux de Y . Ainsi, X peut transférer à Y une information qui va changer les croyances de Y ou bien lui proposer d'adopter un but que Y va transformer en une des ses intentions. Il est donc pertinent de s'intéresser à la façon dont les messages échangés peuvent être structurés pour refléter ces mécanismes d'influence sur les états mentaux des agents.

Les philosophes du langage ont développé au cours des 25 dernières années la théorie des actes de discours qui considère que dire quelque chose c'est en quelque sorte agir. Les paroles sont interprétées comme des actions appelées actes de discours. Cette théorie fournit un cadre qui identifie des types d'actes de discours primitifs et permet de dériver de ces types primitifs l'interprétation sous forme logique de n'importe quel verbe du langage qui exprime un acte de discours quelconque. Les chercheurs en *intelligence*

artificielle ont très tôt pensé à étudier comment les actes de discours peuvent être utilisés pour influencer les états mentaux des agents.

Allen suggéra qu'un locuteur peut identifier des actes de discours directs et indirects s'il est capable de reconnaître le plan de son interlocuteur ainsi que les obstacles qui empêchent sa réalisation. Cependant, cette approche nécessite que l'agent soit capable d'identifier avec précision les catégories d'actes de discours correspondant aux énonciations de ses interlocuteurs, ce qui n'est pas toujours facile comme le remarquent Cohen et Levesque. Ces derniers ont proposé une théorie permettant de tenir compte des états mentaux dans le processus de communication. Ils partent d'une théorie des interactions rationnelles qui se base sur les intentions et engagements des agents. Cette approche qui a influencé grandement les recherches dans ce domaine suppose que les agents sont bienveillants. Galliers a étendu cette théorie à des situations mettant en jeu des agents qui peuvent être en conflit, en montrant qu'il n'est pas nécessaire de considérer que les agents sont bienveillants. Parallèlement Werner a proposé une théorie unifiant des techniques de communication, de coopération et de structuration sociale pour la conception de systèmes d'agents qui adoptent des comportements de groupe : une formalisation des états intentionnels des agents en relation avec les actes de discours et leurs effets sur le processus de planification.

Shoham a proposé une approche formelle décrivant diverses lois sociales applicables aux sociétés d'agents. Castelfranchi et Conte soulèvent diverses objections aux fondements sociologiques actuels de l'IAD : une emphase trop grande est mise sur la coopération tant au niveau des buts partagés, des actions conjointes et du partage d'information; les agents sont *hypercognitifs* dans le sens que l'on dote les agents de capacités d'omniscience, d'introspection qui ne sont pas socialement plausibles; on néglige les mécanismes équivalents à des lois, règlements ou normes qui limitent la liberté des agents; on néglige aussi le rôle de la communication inter-agents comme moyen d'influencer les autres. Ces auteurs proposent une voie alternative dans laquelle on mettrait l'emphase sur les dépendances sociales existant entre agents et sur leur influence dans la formation de buts communs. Pour adopter un comportement social, un agent doit être capable non seulement de raisonner sur ses états mentaux mais aussi de prendre en compte les connaissances qu'il partage avec les autres agents (*mutual belief*) et les engagements qui les relient. On mesure ainsi l'importance d'associer à un modèle de la communication inter-agents un modèle de raisonnement sur les états mentaux, tout en tenant compte de la dimension sociale de l'interaction. Les relations sociales ont aussi un impact évident dans les conversations humaines : la forme et le contenu des actes de discours des locuteurs tiennent compte des rôles et statuts sociaux des participants à la conversation.

Cet aspect n'a été exploré que récemment dans le cadre de la modélisation des interactions et la simulation des conversations entre agents logiciels. Barbuceanu et Fox considèrent une activité de coordination entre agents comme un genre de *conversation* qui est décrite par un graphe d'états finis : les états du graphe représentent les états que peut prendre la conversation. Un ensemble de règles de conversation spécifient comment un agent qui est dans un état conversationnel donné reçoit un message d'un type particulier (spécifié suivant les conventions de KQML), réalise des actions locales (ex. mettre à jour sa base de données locale), envoie des messages et passe dans un autre état conversationnel.

De leur côté, Rousseau et Moulin ont proposé une approche de simulation des conversations entre agents vue comme un processus de négociation multi-niveaux au cours duquel les participants négocient au sujet de leurs états mentaux (croyances, désirs, intentions, émotions, etc.). Dans cette approche les actes de discours sont interprétés par les agents en termes d'objets de conversation (OCs) qui sont des états mentaux associés à des positionnements d'agents. Les OCs proposés par les agents locuteurs au cours de la conversation forment un réseau de concepts qui constitue une mémoire persistante de la conversation partagée par les agents locuteurs et à partir de laquelle ils peuvent raisonner.

Communication utilisant KQML, ACL et les conversations

Le langage KQML a été proposé pour supporter la communication inter-agents. Ce langage définit un ensemble de types de messages (appelés abusivement *performatifs*) et des règles qui définissent les comportements suggérés pour les agents qui reçoivent ces messages. Les types de messages de KQML sont de natures diverses : simples requêtes et assertions (ex. *ask*, *tell*); instructions de routage de l'information (*forward* et *broadcast*), commandes persistantes (*subscribe*, *monitor*); commandes qui permettent aux agents consommateurs de demander à des agents intermédiaires de trouver les agents fournisseurs pertinents (*advertise*, *recommend*, *recruit* and *broker*). Comme le font remarquer Cohen et Levesque, ce langage a été développé de façon ad-hoc pour les besoins des développeurs d'agents logiciels : le terme *performatif* a été utilisé pour nommer diverses commandes qui ont une certaine ressemblance

avec des verbes utilisés de façon performative dans le langage naturel, l'interprétation sémantique actuelle de ces commandes n'est pas satisfaisante.

Ces dernières années, KQML semble perdre du terrain au profit d'un autre langage plus riche sémantiquement ACL (pour Agent Communication Language). Un langage mis en avant par la FIPA qui s'occuppe de standardiser les communications entre agents. ACL est basé également sur la théorie du langage et a bénéficié grandement des résultats de recheche de KQML. Si toutefois, les deux langages se rapprochent au niveau des actes du langage, il n'en ait rien au niveau de la sémantique et il semble qu'un grand soin a été apporté au niveau de ACL tant au niveau de certains protocoles qui sont plus explicites qu'au niveau de la sémantique des actes eux mêmes. Il serait peut-être plus approprié de réviser le langage KQML et ACL en utilisant une sémantique faisant appel à la recursion des actes du langage et un cadre théorique d'interprétation des actes de discours.

3.1.2.6 Méthodes de conception des systèmes multiagents

Les SMAs sont souvent des systèmes complexes qui demandent de longs efforts de développement. Du début des années 80 jusqu'au milieu des années 90, les efforts des chercheurs se sont surtout concentrés sur l'exploration de nouveaux concepts, la mise ne place de théories, la réalisation de divers prototypes de SMAs. Ce bouillonnement d'idées est tout à fait normal dans un jeune domaine technologique: c'est ce que l'on a observé pour les bases de données dans les années 70, les systèmes experts et les systèmes orientés-objets dans les années 80. Cependant, lorsque la technologie atteint plus de maturité et de stabilité, il devient nécessaire de développer des méthodes de conception qui permettent aux entreprises de développer de façon systématique des systèmes opérationnels dans un environnement industriel. Telle est l'évolution que l'on observe pour les SMAs depuis quelques années. Parmi les travaux des pionniers, citons Decker et ses collègues qui ont proposé un cadre de référence pour évaluer les recherches en résolution coopérative de problèmes distribués. Leur but était de comprendre les relations entre des efforts de recherche disparates et d'évaluer les progrès dans ce champ de recherche. Bien que ce cadre ne soit pas une méthode de conception, il soulignait un besoin de prendre du recul par rapport au foisonnement des recherches.

Werner proposa lui aussi un cadre de référence pour essayer de comprendre les relations qui peuvent s'établir entre les usagers, les développeurs et le SMA en construction. Cette réflexion essayait de situer le développement traditionnel de systèmes informatiques par rapport au développement d'un SMA dans lequel les agents pourraient évoluer par eux-mêmes, choisissant leur propres buts, créant leurs propres plans, etc. Ce cadre n'était pas une méthode de conception à proprement parler, mais dénotait une préoccupation méthodologique intéressante. Goodson et Schmidt ont proposé la première démarche méthodologique visant à décomposer un problème multiagents en des unités fonctionnelles, d'identifier les unités fonctionnelles que la machine puisse traiter mieux que les usagers et de construire les modules de résolution de problèmes pour ces unités. Moulin et Cloutier ont développé, quant à eux, la méthode MASB (*Multi-Agent Scenario-Based method*), une méthode de conception de SMA qui se base sur l'analyse et la conception de scénarios caractérisant les rôles joués par les agents humains et les agents artificiels et qui s'applique bien pour des applications de travail collaboratif (ex. système distribué de prise de rendez vous).

Burmeister a proposé une méthode dans laquelle on utilise trois modèles pour analyser un SMA : un modèle d'agent (les agents et leur structure interne définie en termes de croyances, buts, plans, etc.), un modèle organisationnel (description des relations entre agents), un modèle de coopération (description des interactions entre agents). Kinny et ses collègues ont proposé une méthode qui distingue deux niveaux de conception (interne et externe) pour la création d'agents BDI.

Kendall et ses collègues ont proposé, quant à eux, une approche de conception de SMA qui combine des techniques orientées-objet et des techniques de modélisation d'entreprise et de fabrication assistée par ordinateur. Dans cette méthode, les auteurs ont essayé d'adapter les techniques de modélisation des systèmes orientés-objets au domaine des systèmes multiagents (*use cases*, diagrammes de scénarios, patrons de conception, etc.). De leur côté, Maamar et Moulin ont proposé une méthode de conception de systèmes composés d'agents hétérogènes en proposant l'utilisation de cadres (*frameworks*) orientés agents. L'idée est de spécifier les diverses composantes de systèmes distribués et hétérogènes sous la forme d'équipes d'agents dans lesquelles chaque agent joue des rôles spécifiques. Les frameworks, les équipes et les agents sont caractérisés par les services qu'ils rendent et ceux qu'ils requièrent. Diverses techniques de modélisation sont employées pour spécifier les scénarios, les diagrammes d'interactions des agents, les modèles d'agents. Cette méthode a été utilisée pour la réalisation d'un prototype permettant de réaliser

l'interopérabilité de géorépertoires.

La méthode Cassiopea distingue 3 phases pour la conception d'un SMA et a été appliquée au domaine des *robots footballeurs*. Cette méthode utilise des techniques orientées objet pour spécifier les comportements élémentaires des agents. Les comportements relationnels entre agents sont par la suite analysés grâce à un graphe de couplages. Enfin on spécifie la dynamique de l'organisation des agents (qui s'allie avec qui) en utilisant le graphe de couplages. Verharen a, quant à lui, proposé une méthode de conception qui s'appuie sur une perspective de gestion des activités d'un SMA et met notamment en oeuvre les modèles suivants : un modèle d'autorisation qui décrit les types de communications permises et les obligations qui existent entre une organisation et son environnement; un modèle de communication qui raffine le modèle précédent sous la forme de contrats entre les agents sous la forme de réseaux de Pétri, les transactions entre les agents sont modélisées par des diagrammes de transactions qui associent actes de discours et buts des agents, un modèle de tâches qui fournit la décomposition des tâches à partir des diagrammes précédents.

Les méthodes CoMoMAS et MAS-CommonKADS proposent des extensions à la méthode CommonKADS qui a été développée pour le domaine de l'ingénierie des connaissances et en s'appuyant sur certaines techniques orientées-objets (*use cases*, scénarios, modèles de classes). Ces méthodes définissent divers modèles : le modèle d'agent qui décrit les caractéristiques des agents (capacités de raisonnement, buts, services), le modèle de tâches (tâches et sous-tâches), le modèle d'expertise (connaissances nécessaires aux agents pour réaliser leurs tâches), le modèle de coordination (description des interactions possibles entre agents, protocoles et capacités nécessaires), le modèle d'organisation qui décrit comment le SMA va être introduit dans l'organisation qui accueille la société d'agents, le modèle de communication qui décrit les interactions entre les usagers et les agents, le modèle de conception qui synthétise les modèles précédents en tenant compte des contraintes d'implantation. Ces méthodes ont été utilisées dans divers projets de recherche.

Nous pouvons citer également GAIA qui est une méthodologie pour analyser et construire des systèmes à base d'agents. GAIA est appropriée pour le développement de systèmes ayant les caractéristiques suivantes :

- agents gros grains
- maximisation des mesures de qualité globale pouvant être sous-optimal au niveau de chaque élément
- les agents sont hétérogènes (différents langages de programmation et différentes architectures)
- organisation statique (les relations entre les agents et les capacités ne changent pas au cours du temps)

Comme on peut le constater plusieurs méthodes d'analyse et de conception des SMA ont été proposées récemment. La plupart d'entre elles, s'appuient sur des techniques de modélisation empruntées à des méthodes connues en développement orienté-objets ou en ingénierie des connaissances pour aider à la construction des modèles d'agents, de l'architecture du SMA, pour la spécification des modèles d'organisation, d'interaction, etc. Peut-on s'attendre à la création d'une approche standard comme cela fut le cas en ingénierie des connaissances avec la méthode KADS ou à une normalisation des modèles comme ce fut le cas en orienté-objets avec le langage UML? L'avenir nous le dira. Une telle standardisation apparaîtra certainement quand les technologies multiagents seront adoptées effectivement par les entreprises.

3.1.2.7 Quelques exemples d'applications des SMA

De nos jours, la technologie multiagent a trouvé sa place dans les systèmes manufacturiers, les systèmes financiers, les loisirs, les télécommunications, le contrôle-commande, les systèmes embarqués, et pas mal d'autres applications.

Dans ce qui va suivre nous n'en exposerons que quelques exemples d'applications utilisant cette technologie.

Application des SMA aux télécommunications

Ces dernières années, les télécommunications ont notamment introduit une conception de services décentralisée dans le contexte du Web, créé de nouveaux services de médiation tels que les portails et

engendrés l'apparition de nombreux fournisseurs de services réseaux qui ne disposent pas de leurs propres services réseaux. L'obtention de tels services décentralisés ne peut, bien entendu, être obtenue que grâce à des logiciels pour lesquels les données et le contrôle sont forcément distribués. De ce fait, il est clair que les SMAs semblent convenir aux télécommunications. C'est pourquoi les principaux acteurs de télécommunications mènent actuellement d'intenses activités de recherche sur la technologie agent : British Telecom, France Télécom, Deutch Telekom, NTT, Nortel, Siemens, etc.

Le système ADEPT

Les gestionnaires de grandes compagnies effectuent des prises de décisions en se basant sur une combinaison de jugement et d'informations provenant de plusieurs départements. Idéalement, toutes les informations pertinentes devraient être rassemblées avant qu'une décision ne soit prise. Cependant, le processus d'obtention des informations, à jour et pertinentes, est très complexe et prend énormément de temps. Pour cette raison, plusieurs compagnies ont cherché à développer des systèmes informatiques afin de les assister dans leur processus d'affaires.

Le système ADEPT attaque ce problème en voyant le processus d'affaires comme un ensemble d'agents qui négocient et qui offrent des services. Chaque agent représente un rôle distinct ou un département de l'entreprise et est en mesure de fournir un ou plusieurs services. Les agents qui requièrent les services d'autres agents le font par une négociation qui permet d'obtenir un coût, un délai temporel et un degré de qualité qui sont acceptables aux deux parties. Le résultat d'une négociation terminée avec succès constitue un engagement entre les deux parties.

Le système GUARDIAN

Le système GUARDIAN a pour but de gérer les soins aux patients d'une unité de soins intensifs chirurgicale. Les principales motivations de ce système sont: premièrement, le modèle des soins d'un patient dans une unité de soins intensifs est essentiellement celui d'une équipe, où un ensemble d'experts dans des domaines distincts coopèrent pour organiser les soins des patients; deuxièmement, le facteur le plus important pour donner de bons soins au patients est le partage d'informations entre les membres de l'équipe de soins critiques. Particulièrement, les médecins spécialistes n'ont pas l'opportunité de superviser l'état d'un patient minute par minute, cette tâche revient aux infirmières qui, quant à elles, ne possèdent pas les connaissances nécessaires à l'interprétation des données qu'elles rassemblent.

Le système GUARDIAN répartit donc le suivi des patients à un certain nombre d'agents de trois types différents. Les agents perception/action sont responsables de l'interface entre GUARDIAN et le monde environnant, établissant la relation entre les données des senseurs et une représentation symbolique que le système pourra utiliser, et traduisant les requêtes d'action du système en commandes pour les effecteurs. Les agents en charge du raisonnement sont responsables d'organiser le processus de prise de décision du système. Finalement, les agents en charge du contrôle (il n'y en a habituellement qu'un seul) assurent le contrôle de haut niveau du système.

Les systèmes d'informations coopératifs (SIC)

Les SIC sont généralement caractérisés par la grande variété et le grand nombre de sources d'informations. Ces sources d'informations sont hétérogènes et distribuées soit sur un réseau local (Intranet) soit sur l'Internet. De tels systèmes doivent être capables d'exécuter principalement les tâches suivantes : découverte des souces, recherche d'informations, filtrage des informations, fusion des informations.

Le système multiagent Warren pourrait constituer un exemple spécifique de l'utilisation des agents dans ce type d'application. C'est un système d'agents intelligents pour l'aide des usagers dans la gestion des portefeuilles. Ce système combine les données du marché financier, les rapports financiers, les modèles techniques et les rapports analytiques avec les prix courants des actions des compagnies. Toutes ces informations sont déjà disponibles sur le Web, Warren ne fait que les intégrer via des agents spécialisés, les agents d'informations et ensuite les présenter aux usagers.

Il existe bien d'autres exemples d'applications.

3.2 Les plate-formes multiagents

L'objet du stage étant de déployer une bibliothèque de déploiement d'agents, nous allons dresser ici un état de l'art du domaine qui nous permettra de comprendre comment se situe les développements qui ont fait l'objet du stage. Nous ne prétendons nullement ici à l'exhaustivité, le but étant d'avoir un aperçu superficiel de ce qui a été fait.

L'émergence des systèmes multiagents a donné lieu à l'élaboration de plusieurs méthodologies et architectures pour la modélisation des SMA. Le concept de programmation orientée-agent est une idée intéressante et les méthodologies développées fournissent des patrons théoriques pour la modélisation des SMA. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont souvent difficiles à implémenter directement à partir des langages de programmation standards comme Java ou C++.

Plusieurs outils (éléments logiciels offrant des services pour le développement de SMA) de différents types ont été développés récemment pour la programmation orientée-agent.

3.2.1 Critères d'évaluation des outils

Voici quelques caractéristiques qui sont fréquemment mentionnées pour évaluer les outils de développement des SMA :

- support du déploiement multimachines
- diminution de l'effort nécessaire à l'implémentation
- abstraction assez forte pour permettre à néophyte de créer un SMA sans se soucier des détails implémentatoires
- code facilement extensible
- les développeurs n'ont pas à se soucier de l'implémentation du système de communication et des protocoles utilisés pour le transport des messages
- outils de débogage
- interface utilisateur facilitant le développement

3.2.2 Quelques outils

AgentTool

Cet outil se base sur une méthodologie qui se veut une extension au modèle OO : la méthodologie MaSE. Celle-ci comporte sept phases : trouver les buts, appliquer les cas d'utilisation, raffiner les buts, créer les classes d'agents, construire les conversations, assembler les classes d'agents et l'implémentation. Cette méthode met l'accent sur l'analyse et le développement. L'outil permet la vérification et la validation des conversations. Le déploiement (partiel) se fait directement à l'intérieur de l'environnement. La génération automatique du code (en Java) des conversations est disponible. Cet outil est intéressant pour effectuer les premières étapes du développement d'un SMA.

AgentBuilder

AgentBuilder est un environnement de développement complet. Une modélisation orientée-objet constitue la base de la conception des systèmes à laquelle on ajoute une partie ontologie. L'élaboration du comportement des agents se fait à partir du modèle BDI. KQML est utilisé comme langage de communication entre les agents. L'exécution du système se fait à partir du moteur d'exécution d'AgentBuilder. Par contre, on peut créer des fichiers *.class* et les exécuter sur une JVM standard. AgentBuilder est un outil complexe qui demande des efforts d'apprentissage importants et de bonnes connaissances dans le domaine des systèmes multiagents pour être utilisé de façon performante. Il est limité au niveau de l'extensibilité, du déploiement et de la réutilisabilité.

Ciao Prolog

Fonctionnalités de Ciao Prolog (M. Hermenegildo et al 1999) :

- Threads, multi-machine

- Communication et synchronisation par une base de faits Prolog
- Règles conditions-actions (extensions syntaxique de Prolog)
- Objets
- Objets actifs (démons répondant aux requêtes à distance)

Decaf

Decaf est un environnement de développement de plans. L'outil fourni quelques utilitaires pour l'élaboration de plans et pour la coordination des tâches. Un planificateur applique des heuristiques pour trouver un ordonnancement aux tâches. Une interface permet la construction de celles-ci. DECAF fourni aussi un éditeur d'agent qui est utile pour le débogage. Aucune méthodologie n'est spécifiée pour la conception.

Eel

Fonctionnalités de Eel (T.S. Dahl 1999) :

- Processus communiquants
- Communication synchrone point à point (unification de terme)

Gaea

Fonctionnalités de Gaea (I. Noda, H. Nakashima, K. Handa 1999) :

- Threads, multi-machines
- Communication par mémoire partagée

Jack

L'environnement Jack est constitué d'un éditeur gestionnaire de projet, d'un langage de programmation JAL (Jack Agent Language) et d'un compilateur. Le gestionnaire de projet est une interface qui possède un éditeur de textes où se fait l'implémentation du système. La compilation (passage de JAL à Java) et l'exécution du système se font aussi à l'intérieur de cette interface. Le langage JAL est une extension à Java. Aucune méthodologie n'est proposée. Les agents sont basés sur un modèle BDI. Aucun éditeur n'est disponible pour le développement ou le déploiement des systèmes. Jack est très long à maîtriser, il faut apprendre le langage JAL et connaître le modèle BDI. De plus, le manque de support graphique complique l'implémentation et le déploiement des systèmes.

Jade

Jade est un outil qui répond aux normes FIPA97. Aucune méthodologie n'est spécifiée pour le développement. Jade fourni des classes pour la définition du comportement des agents. L'outil possède trois modules principaux (nécessaire aux normes FIPA). Le DF *director facilitor* fourni un service de pages jaunes à la plateforme. Le ACC *agent communication chanel* gère la communication entre les agents. Le AMS *agent management system* supervise l'enregistrement des agents, leur authentification, leur accès et utilisation du système. Les agents communiquent par le langage FIPA ACL. Un éditeur est disponible pour l'enregistrement et la gestion des agents. Aucune autre interface n'est disponible pour le développement ou l'implémentation. À cause de cette lacune, l'implémentation demande beaucoup d'efforts. Elle nécessite une bonne connaissance des classes et des différents services offerts.

JAFMAS et JiVE

JAFMAS met l'emphase sur les protocoles de communications, l'interaction entre les agents, la coordination et la cohérence à l'intérieur du système. Il propose une méthodologie en cinq phases : identifier les agents, identifier les conversations, identifier les règles de conversation, analyser le modèle des conversations et l'implémentation. L'éditeur graphique (JiVE) est un outil de support pour le développement qui propose une interface qui aide l'utilisateur dans sa démarche. Une particularité de JiVE est la possibilité de travailler en groupe sur un projet. Les réseaux de Pétri rendent la création de conversations et la

coordination complexes. Aucun support pour le déploiement n'est disponible.

Jinni (Java INference Engine and Networked Interactor)

Voici les principales fonctionnalités offertes par Jinni (P. Tarau 1999) :

- Intégration Prolog-Java
- Threads, multi-machine, mobilité
- Synchronisation par unification de termes
- *blackboard* pour la communication entre les threads

MadKit

Madkit est un environnement basé sur la méthodologie Aalaadin ou AGR (agent / groupe / rôle). L'outil fourni un éditeur permettant le déploiement et la gestion des SMA (G-box). La gestion faite via cet éditeur offre plusieurs possibilités intéressantes. L'outil offre aussi un utilitaire pour effectuer des simulations.

Qu Prolog

Qu-Prolog (K.L. Clarck, P Robinson, R. Hagen 1998) est un Prolog étendu. Il est *multi-threads* et offre des communications évoluées entre les threads, les processus et les machines. Il s'agit donc d'un outil bien adapté et puissant pour la programmation d'agents. Qu-Prolog aurait pu constituer une solution intéressante pour le service, il semble techniquement satisfaisant mais a un caractère industriel insuffisant.

Zeus

Zeus est un environnement complet qui utilise une méthodologie appelée *role modeling* pour le développement de systèmes collaboratifs. Les agents possèdent trois couches. La première couche est celle de la définition où l'agent est vu comme une entité autonome capable de raisonner en termes de ses croyances, ses ressources et de ses préférences. La seconde couche est celle de l'organisation. Dans celle-ci, il faut déterminer les relations entre les agents. La dernière couche est celle de la coordination. Dans celle-ci, on décide des modes de communication entre les agents, protocoles, coordination et autres mécanismes d'interactions. L'outil est un des plus complets. Les différentes étapes du développement se font à l'intérieur de plusieurs éditeurs : ontologie, description des tâches, organisation, définition des agents, coordination, faits et variables ainsi que les contraintes. Le développement de SMA avec Zeus est cependant conditionnel à l'utilisation de l'approche *role modeling*. L'outil est assez complexe et sa maîtrise nécessite beaucoup de temps.

3.2.3 Bilan

La majorité des outils développés le furent pour exploiter ou démontrer un concept ou une idée en particulier. De ce fait, le développement de ces outils négligent, volontairement ou non, le développement de plusieurs dimensions essentielles à l'implémentation d'un SMA. Ceci rend leur utilisation inappropriée ou même impossible pour le développement de systèmes réels. Un constat s'impose : les outils existants ne permettent pas encore le développement complet, et ce de façon relativement simple, de SMA appliqués à des systèmes réels d'envergure intéressante.

3.3 Les systèmes multiagents dans le service

L'*Intelligence Artificielle Distribuée*, domaine dans lequel s'inscrivent les systèmes multiagents, représente une voie active de recherche. Dans ce contexte foisonnant d'idées et de vocables différents, dont l'usage a malheureusement parfois été fortement galvaudé, il nous a semblé souhaitable d'exposer de manière claire et précise les raisons qui ont poussé à investiguer davantage dans cette voie et surtout de bien définir les termes et les grands concepts qui gouvernent l'élaboration de ces systèmes. Nous nous sommes permis ici de reprendre certaines parties du rapport d'activités 2003-2004 du service car il nous a semblé qu'une reformulation se serait faite au détriment de la clarté et de la concision du propos.

3.3.1 Cadre d'utilisation

« La raison profonde nous ayant conduits à investiguer la voie des systèmes d'agents est de lutter contre la complexité structurelle des programmes reprenant ainsi les principes de modularité ayant guidé l'informatique depuis sa création. Ce qui change ici c'est la nature du module : l'agent. A la différence des approches successives de la modularité (la procédure, l'objet, la tâche pour n'en citer que les principales), la caractérisation d'un agent reste encore aujourd'hui un sujet de dicussion et force est de constater que, suivant les domaines d'application, les agents que l'on y rencontre peuvent correspondre à des réalités bien différentes. Les systèmes de mission qui nous guident n'échappent pas à la règle et c'est pourquoi nous avons dû dans un premier temps définir avec précision le concept d'agent correspondant à nos attentes. »

3.3.2 Caractérisation des agents

« Les systèmes qui nous préoccupent ne sont pas, à la différence du courant dominant que représentent le commerce électronique ou les services Web, des systèmes ouverts, c'est à dire développés indépendamment les uns des autres, par des organismes différents pas forcément coopératifs. L'agent est un moyen pour rendre plus intelligent nos programmes sans accroître leur complexité structurelle. C'est la raison pour laquelle nous ne nous sommes pas intéressés aux nombreux travaux de normalisation de ces dernières années (tels ceux conduits par la FIPA) mais au contraire aux propriétés intrinsèques des agents.

Nous avons finalement dégagé une définition de l'agent qui s'appuie essentiellement sur la notion d'autonomie et de dépendance limitée que nous définissons ci-dessous.

3.3.2.1 Principe d'autonomie

Il s'énonce ainsi :

> **Un agent A est autonome par rapport à B si B ne peut pas prédire à coup sûr le comportement de A.**

Que l'on ne s'y trompe pas, malgré son apparence anodine, une telle exigence a des conséquences extrêmement fortes sur la façon de concevoir un système, tant sur sa décomposition en agents que sur la définition du comportement individuel de chacun d'entre eux. Accepter une part d'indétermination dans le comportement des programmes ne fait pas partie des pratiques actuelles de la conception des applications de mission, c'est le moins que l'on puisse dire puisque ; au contraire, tout est mis en place pour offrir la meilleure et la plus fine spécification possible afin d'éviter tout aléa dans la conception de chaque module.

Lorsque nous avons « découvert » l'autonomie sous cet angle, nous avons tout d'abord pensé qu'une telle méthodologie ne s'appliquait pas aux systèmes de mission qui nous intéressent. Mais les expérimentations et les réflexions sur la tolérance aux fautes que nous avons menées nous ont permis de surmonter nos réticences initiales pour prendre conscience que donner de l'autonomie aux agents avait des conséquences tout à fait positives, parmi lesquelles :

- les agents ne pourront pas ignorer le blocage possible des autres agents (en particulier si ce blocage résulte de la mise en œuvre d'un algorithme de recherche heuristique dont on ne maîtrise pas la complexité combinatoire,
- les agents devront anticiper la perte possible d'un message puisque, du fait de leur autonomie, les autres agents pourront décider de ne pas envoyer une réponse normalement prévue.

Ainsi donc le principe d'autonomie, tout en guidant le concepteur vers une structuration du système plus apte à prendre en compte des algorithmes complexes tels ceux qu'a produit l'*Intelligence Artificielle*, permet également une prise en compte dès la conception de la tolérance aux fautes.

3.3.2.2 Principe de la dépendance limitée

Ce principe est une conséquence quasi inéluctable du principe d'autonomie ou, plus précisément, il est nécessaire que les agents aient le minimum de dépendances pour que le principe d'autonomie puisse

être mis en pratique.

En conséquence, le seul mécanisme reliant les agents les uns aux autres est celui de la passation de message. Aucune autre ressource (en particulier la mémoire) ni aucun autre moyen de synchronisation (par exemple le sémaphore) ne sera à la disposition des agents, sauf à mettre en place un troisième agent chargé de la médiation.

La mise en pratique de ces deux principes fournit au concepteur une vision complètement différente de ce qu'offrent les approches traditionnelles mais surtout cela le place dans un environnement simplifié lui offrant ainsi un espace de liberté qu'il pourra utiliser pour la mise en œuvre des algorithmes complexes dont nous avons fait état précédemment. Que la tolérance aux fautes trouve également ici un cadre pour pouvoir (et même devoir) être prise en compte « à la source » est une incitation supplémentaire à développer ce type d'approche. »

3.3.3 Programmation des agents

3.3.3.1 Présentation du langage Prolog

Prolog est un langage de programmation qui fut développé dans le courant des années 70-75 à l'université de Marseille. A la différence de tous les autres langages existants à cette époque, il permet l'écriture de programmes décrits comme un ensemble de règles de la logique des prédicats du premier ordre, d'où son appartenance à la classe des langages de programmation logique.

Unification et retour-arrière (en anglais « backtracking ») en sont les deux mécanismes principaux.

Unification

Le concept d'**unification** est une notion centrale de la logique des prédicats ainsi que d'autres systèmes de logique et est sans doute ce qui distingue le plus Prolog des autres langages de programmation.

L'unification de deux termes *t1* et *t2* consiste à trouver (quand il en existe) un troisième terme *t* tel qu'on puisse passer de *t* à *t1* et à *t2* en instanciant certaines variables. *t* est alors appelé un *unificateur* de *t1* et *t2*. Intuitivement, l'unification est le fait d'attribuer une valeur à certaines variables de t1 et t2 et peut être regardé comme un genre d'assignation qui ne pourrait s'effectuer qu'une seule fois. Lorsqu'on résout une équation algébrique, une inconnue peut avoir une, plusieurs ou aucune solution, mais sa valeur ne change pas durant les opérations ; c'est pareil pour l'unification. En fait on peut voir la résolution d'une équation comme un cas particulier d'unification.

Backtracking

Le **retour sur trace**, appelé aussi ***backtracking*** en anglais, est une stratégie pour trouver des solutions aux problèmes en utilisant un langage de programmation déclaratif, comme Prolog et dans d'autres situations comme la décomposition analytique de texte. L'idée maîtresse est d'essayer chaque possibilité jusqu'à trouver la bonne. C'est une recherche en profondeur d'abord sur l'ensemble des solutions.

Pendant la recherche, si vous essayez une alternative qui ne fonctionne pas, vous faite un retour sur trace au point choisi où se présente différentes alternatives, et vous essayez la possibilité suivante. Si vous n'avez plus de points la recherche échoue.

Il est possible de mettre en place une **coupure**, symbolisée par un '**!**'. La coupure est un prédicat sans signification logique (la coupure n'est ni vraie ni fausse), utilisé pour « couper » des branches de l'arbre de recherche.

Il existe de nombreuses implémentations de ce langage, dont celle qui est utilisée dans le service à savoir SICStus Prolog dont le choix nous le verrons n'a pas eu une influence anodine.

Le langage est, du point de vue de l'utilisateur, clair, lisible, concis, et permet une grande rapidité d'écriture.

C'est un langage particulièrement adapté aux systèmes sur les langages naturels, les systèmes experts, en fait à de nombreux domaines de l'*Intelligence Artificielle*. En effet, les données de base manipulées par Prolog sont les arbres et plus généralement les graphes, données omniprésentes en IA notamment pour la représentation des connaissances. Un nombre illimité de types peuvent être utilisés, sans avoir à être déclarés séparément. Pour manipuler de telles données Lisp ou Scheme utilise les notions de constructeur et de destructeur, Prolog quant à lui fournit un mécanisme plus général : le pattern-matching utilisé dans le cadre de l'unification.

Un calcul en Prolog peut être vu comme une suite d'appels dirigés de procédures. Du fait de l'importance des « procédures » il est nécessaire d'avoir plus de flexibilité que dans les autres langages, l'utilisation d'une procédure peut varier d'un appel à l'autre, en effet les paramètres peuvent suivant le cas être :

- des paramètres de retour (résultats)
- des paramètres d'entrée (données) et cela n'a pas besoin d'être déterminé préalablement, cette propriété permet aux procédures d'être multi-directionnelles.

De ce fait lorsque nous écrirons des prédicats Prolog nous utiliserons la convention suivante :

- +Nom : Nom doit être instancié à l'appel
- -Nom : Nom doit être libre à l'appel
- ?Nom : Nom peut être libre ou lié

3.3.3.2 De l'usage de la programmation logique dans le service

L'utilisation du langage Prolog était un impératif pour les développements du stage. Nous allons expliciter ici les raisons qui justifient ce choix.

Il peut sembler étonnant que la première exigence mise en avant concerne le langage de programmation des agents à l'heure où les pratiques consistent plus à choisir, et cela le plus tard possible, le langage le mieux adapté à la tâche à réaliser. Il s'avère que cette attitude est pertinente lorsqu'il s'agit du développement de logiciels mettant en œuvre des algorithmes déterminisites en général parfaitement connus tant en ce ce qui concerne leur fonctionnalité que leur temps d'exécution ou occupation mémoire.

Par contre, dès qu'il s'agit de réaliser des fonctions nécessitant le parcours d'un espace de recherche très vaste, ce qui est la caractéristique première des programmes issus de l'*Intelligence Artificielle*, de développer des algorithmes mettant en œuvre des techniques de raisonnement hypothétique, de planification dans un univers incertain, etc. De plus, Prolog possède en son cœur un double mécanisme d'unification (analogue à celui que l'on met en jeu lorsque l'on décide d'appliquer un théorème en mathématique) et de résolution (recherche en profondeur d'abord) qui a fait ses preuves dans bien des développements.

Les langages traditionnels s'avèrent très mal adaptés en particulier lorsqu'il s'agit de créer rapidement des prototypes pour évaluation des solutions retenues. L'un des intérêts majeurs de Prolog réside donc dans sa productivité, entendu au sens de capacité à développer rapidement des programmes avec de faibles moyens. Cette productivité est évidemment indispensable dans une optique de recherche qui oblige à effectuer des allers retours incessants entre conception et expérimentation. Mais ce qui est également exceptionnel c'est que cette aptitude à programmer rapidement des programmes complexes ne se fait en aucun cas au détriment de leur lisibilité, de leur maintenabilité ni même de leur efficacité. De nombreux exemples développés en interne peuvent en attester.

Il faut ajouter à ce tableau des capacités d'introspection et de modifications dynamiques du code source qui peuvent, bien maîtrisées, se montrer extrêmement puissants.

Aujourd'hui, le langage Prolog ne suscite plus le même engouement que par le passé mais, outre le fait que de nombreuses équipes industrielles continuent de l'utiliser même si elles n'en font pas état (Microsoft et Apple n'en étant pas les moindres exemples), Prolog a acquis aujourd'hui une maturité qui fait de lui un langage utilisable dans la plupart des contextes opérationnels. Le fait que Prolog ait été à l'origine du développement de la programmation à contraintes ne fait que confirmer la place particulière qu'il occupe dans l'histoire des langages informatiques.

Le langage Prolog semble donc être le meilleur candidat et il est largement utilisé dans le groupe depuis une vingtaine d'années. Par ailleurs, le service dispose d'un corpus de programme important, en particulier dans le domaine de la programmation à contraintes susceptibles de jouer un rôle important lors de la programmation de la composante intelligente des agents.

Il était donc important dans cette optique de disposer d'un environnement adapté à la programmation d'agents en Prolog.

3.3.4 L'existant

Nous allons dresser ici un état des lieux des outils présentant un rapport, plus ou moins direct, avec le sujet du stage, dans l'état où ils se présentaient au mois de Janvier 2005. Il a bien évidemment été nécessaire de se familiariser avec chacun d'eux afin d'acquérir une vue d'ensemble indispensable aux développements du stage.

3.3.4.1 Langages et outils utilisés

Présentons brièvement les langages et outils qui sont utilisés au sein des applications du service.

SICStus Prolog

Comme nous l'avons mentionné, le langage Prolog est au cœur des développements de la division. Nous ne reviendrons pas sur ce point acquis. Il convient par contre de présenter succintement l'implémentation de Prolog qui fut utilisée à savoir SICStus Prolog.

SICStus Prolog est une implémentation commercialisée par un institut suédois (SICS). Elle a été choisie pour plusieurs raisons :

- respect des standards ISO
- disponibilité sur de nombreuses architectures (Intel/x86, Alpha, Compaq, Sun Sparc, ...) et systèmes d'exploitation (Windows, Linux, Solaris...)
- excellente gestion de la mémoire
- excellentes performances
- génération de code natif
- debugger avancé
- interface bidirectionnelle avec C/C++, Java, Tcl/Tk...
- nombreux modules mis à la disposition du programmeur

SICStus Prolog est donc une implémentation extrêmement complète, de qualité industrielle et qui a déjà fait ses preuves dans de nombreux domaines (Aérospatiale, Télécommunications...).

Java

Le langage Prolog n'étant guère adapté à l'élaboration d'interfaces graphiques ou à l'implémentation de certains algorithmes, le langage Java est communément utilisé par le service pour tout ce qui est visuel. Se référer à la partie sur NAOMI pour mieux comprendre comment cela se déroule en pratique.

Microsoft Batch Scripting Language

Le langage de script de Microsoft est utilisé ponctuellement afin d'initialiser des variables d'environnement et d'appeler les points d'entrée de nos applications.

3.3.4.2 Bibliothèque de déploiement (ALBA)

Description[1]

La mise en œuvre effective d'applications multiagents nécessite un environnement de développement et de déploiement. Le premier réflexe fut de recourir à une plate-forme[2] du « commerce » ; il en existe de nombreuses, pour beaucoup en accès libre. Trois raisons ont conduit au développement d'une plate-forme spécifique :

- les modèles d'agents mis en œuvre dans les plate-formes disponibles ne correspondent pas nécessairement à ce qui est recherché, principalement parce que nous sommes encore peu nombreux à explorer cette voie pour des applications telles que les nôtres,
- le but est d'être en mesure de valider les idées dans un contexte opérationnel et donc pour cela il ne faut pas dépendre trop fortement de solutions fermées,
- le service dispose d'un acquis important construit sur Prolog et peu de plate-formes le supportent

Par ailleurs, il nous est rapidement apparu qu'un tel développement pouvait se faire avec des moyens relativement limités et donc que la liberté de manœuvre qui en résultait rentabiliserait rapidement l'investissement consenti. C'est ce qui a pu être observé en fin de compte.

ALBA est un environnement pour la construction d'agents programmés en Prolog ne présupposant pas du modèle d'agent utilisé (ce qui nous permet d'en mettre en œuvre autant que nécessaire en fonction des problèmes à résoudre). Elle permet le déploiement sur plusieurs machines ainsi que la migration d'une machine à l'autre.

La « plate-forme » ALBA est complètement décentralisée (à la différence de beaucoup d'autres). Cela signifie que le code nécessaire à l'exécution et à la communication est « embarqué » dans chaque agent et qu'aucun mécanisme central n'est mis en œuvre. Ce point également est important pour la tolérance aux fautes. ALBA a été utilisée pour les différents démonstrateurs que nous avons réalisés. Nous sommes en mesure de nous intégrer dans la plupart des environnements existants.

ALBA 1.0

Description

Après avoir tenté de développer une plate-forme en Java, dénommé Ares, c'est dans le contexte précédemment décrit que Mlle Caroline Chopinaud a développé la bibliothèque ALBA 1.0. Ares fut en effet jugée trop lourde, peu évolutive, elle présentait, d'autre part, des communications inefficaces (conversion Prolog vers Java, Java vers Prolog) ainsi qu'une centralisation inutile et indésirable des données.

La bibliothèque Alba 1.0 a donc été mise en chantier et achevée au printemps 2004. Elle présente l'ensemble des fonctionnalités classiques que l'on est en droit d'attendre d'un tel outil :

- création d'agents sur machine locale et distante
- envoi et réception de messages
- arrêt d'un agent
- migration

Critiques

Bien que fonctionnant, somme toute, de manière plus qu'honorable, on se doute que la version 1.0 d'ALBA n'était pas exempte de défauts sinon la bibliothèque n'aurait évidemment pas fait l'objet du présent stage. Nous allons donc nous attacher ici à décrire brièvement le fonctionnement d'ALBA 1.0 et d'en comprendre les limitations.

Il faut tout d'abord noter que dans cette mouture, tous les prédicats offerts au programmeur d'agents prennent en dernier paramètre une variable intitulée *KL* pour *Knowledge*. Il s'agit d'une représentation interne à ALBA des données utiles à son exécution. Tous les agents du système sont identifiés par une *IDCARD* contenant le nom de l'agent, son adresse (adresse IP de la machine sur laquelle il réside) et le numéro de port sur lequel il écoute. Ces informations sont stockées sous la forme de variables dites « *mutables* ».

1 Issu du rapport d'activité 2003-2004
2 Comme nous l'avons dit en introduction le terme de plate-forme n'est pas forcément très adapté

Variable Mutable
Les variables « *mutables* », sont des variables Prolog particulières dans le sens où elles sont à assignations multiples (les variables classiques ne sont instanciables qu'une seule fois) tout en n'étant pas résistantes aux « *backtrack* » (Cf. présentation du langage Prolog). Ces variables sont associées en interne à un historique des instanciations couplé à des indications temporelles qui permettent en quelque sorte de revenir dans le passé et de défaire ce qui a été fait.

L'utilisation de variables « *mutables* » n'apparaît pas souhaitable dans le cadre d'une bibliothèque multiagent notamment pour ce qui est de stocker les informations relatives à l'identification des agents, il est par exemple absurde de revenir sur la création d'un agent, on ne peut défaire l'envoi d'un message... Cette constatation s'est vérifiée en pratique puisque, de part ce choix technique, la migration des agents ne fonctionne pas correctement dans la première mouture de la bibliothèque. Il semble donc pertinent d'utiliser des procédés permettant un stockage persistant des données internes d'ALBA.

Une deuxième critique fut également formulée en terme d'interface avec le programmeur. ALBA 1.0 faisait en effet remonter trop de messages internes aux programmeurs d'agents. Ces messages complexifiaient inutilement la programmation du comportement des agents, forçant le programmeur à se soucier de préoccupations de bas niveaux[1], alors qu'il doit se placer à un niveau d'abstraction bien supérieur et être libéré de ce genre de contingences.

On peut également reprocher à ALBA 1.0 un certain manque de maturité et une gestion parfois incomplète des erreurs susceptibles de se produire lors de l'exécution d'un SMA. Nous verrons, enfin, que la gestion des messages était perfectible.

La première mouture de la bibliothèque ALBA a donc constitué un excellent banc d'essai qui a permis de valider un certain nombre de concepts de la vision des SMA du service. Elle a notamment permis de mettre en exergue la souplesse et la puissance que pouvait apporter le Prolog pour un tel développement. Elle a d'autre part offert à l'équipe la possibilité de capitaliser une bonne expérience sur la question qui a autorisé la conduite du développement qui a fait l'objet de mon stage. L'objectif était donc de tirer le meilleur parti de ces acquis afin de pallier les limitations d'ALBA 1.0 et de proposer toutes les améliorations nous semblant pertinentes.

3.3.4.3 Comportement des agents

Décrire le comportement des agents est une tâche difficile qui fait l'objet de recherches actuellement. L'équipe a donc développé différents outils permettant de simplifier le développement des agents. Ces travaux sont encore relativement jeunes et sont donc sujets à de profondes évolutions.

Comportement séquentiel

Pour certains agents extrêmements naïfs, il peut être suffisant d'en programmer le comportement de manière complètement linéaire en utilisant une suite monolithique d'appels à des routines Prolog dont notamment des appels directs aux routines de communication d'ALBA.

Les automates

Des agents un peu plus évolués peuvent être programmés en utilisant une bibliothèque d'automates Prolog qui a été conçu par le service. Il s'agit alors de décrire le comportement de l'agent comme une suite d'état dont les transitions sont gouvernées par les messages reçus.

Les fils de raisonnement

Le comportement d'un agent est orienté par les messages qu'il reçoit de l'extérieur. Si l'on essaye de faire une analogie[2] avec le fonctionnement que l'on adopte en lisant notre courrier, on se rend compte que nous avons tendance dans un premier temps à le trier. Nous sommes ensuite capable de mettre en place un

1 Problèmes de communications, de connexion...
2 Nous verrons qu'il est souvent intéressant d'avoir recours à des métaphores issues de la vie quotidienne pour résoudre certains problèmes pratiques

raisonnement local qui nous permet de traiter chaque missive, d'utiliser notre mémoire afin de relier les lettres qui ont un rapport entre elles, d'ignorer les publicités ou les messages de peu d'intérêt... C'est avec cette vision à l'esprit, que les fils de raisonnement ont été mis en place par M. Patrick Taillibert.

Il s'agit d'une extension des automates. Chaque fil de raisonnement peut-être vu comme un contexte. Il est possible de créer et supprimer dynamiquement des fils de raisonnement en fonction de l'évolution du système. Chaque fil de raisonnement présente une mémoire locale et se décline, pour l'instant, sous la forme d'un automate. Une mémoire globale à tous les fils de raisonnement permet un partage d'informations entre ces derniers. Les messages sont lus au niveau d'un point central, le « *switch* » qui se charge grâce à des règles de grammaire (pouvant travailler sur le contenu du message, l'expéditeur, etc) de filtrer les messages et de les envoyer aux fils de raisonnement appropriés. Il est bien sûr possible d'ajouter et de supprimer des règles de grammaire à l'exécution. Les fils de raisonnement prendront encore plus de puissance lorsqu'ils seront couplés aux outils de génération de plan qui permettront une mise en place automatique et dynamique du comportement des agents.

Cet outil facilite grandement le travail du programmeur qui n'a plus qu'à se concentrer sur un comportement local de l'agent lié à un contexte précis, ce qui réduit grandement la complexité des agents et accroît la lisibilité de leur code. Il est à noter que les fils de raisonnement, qui ont déjà fait grandement leur preuve en pratique, encapsulent les routines de communication d'ALBA qui deviennent invisibles pour le programmeur. Il est envisagé de le faire évoluer vers un langage de plus haut niveau. Des recherches sont en outre menées sur l'organisation des mémoires, bien que l'outil reste tourné vers la généricité.

Afin d'avoir un meilleur aperçu des automates et des fils de raisonnement le lecteur pourra se référer au rapport de François Chazal qui a décrit la programmation du comportement des agents de façon plus exhaustive.

3.3.4.4 NAOMI

Motivation

La prise en compte des Intefaces Homme Machine et son incorporation dans la vision SMA du service, soulève le problème technique suivant : sachant que le coeur cognitif d'un agent est conçu en Prolog, et que, en contrepartie, Java semble une bonne solution pour l'affichage graphique, comment coupler efficacement les deux, en respectant les contraintes propres à l'implémentation au niveau SMA. Bien que SICStus Prolog fournisse des outils[1] pour interfacer du code Prolog et du code Java, ils n'étaient pas pleinement satisfaisant pour l'usage que l'on voulait en faire, c'est ce qui a motivé le développement de la plate-forme NAOMI.

Fonctionnement

Le package NAOMI fournit une plate-forme permettant le contrôle de code Java depuis un (ou plusieurs) programme Prolog et cela dans l'optique de pouvoir doter facilement des agents, écrits en Prolog, d'une interface utilisateur, écrite en Java.

Ce contrôle inclut l'invocation de code Java (instanciation de classe, appel de méthodes) depuis Prolog, ainsi que la communication de résultats éventuels (pour une fonction) ou d'évènements particuliers (activation d'une option, fermeture d'une fenêtre,...) par le code Java.

D'un point de vue technique, la plateforme est basé sur les principes suivants :

- communication réseau : TCP/IP avec design client-serveur classique
- support de clients Prolog multiples, chacun ayant une unité de traitement dédiée
- une unique JVM pour exécution de code Java, permettant de factoriser et de centraliser l'exécution de code Java au sein d'une entité unique
- Utilisation des principes de réfléxivité propres à Java pour l'invocation et la manipulation de code au runtime
- application multi-thread, permettant notamment le contrôle et la supervision des clients Prolog et de leur pendant Java

1 PrologBeans et Jasper

3.3.4.5 SpySMA, Phenix

L'impact de plus en plus important des systèmes multiagents dans le développement des applications de l'*Intelligence Artificielle* réalisées chez Thales Systèmes Aéroportés, a obligé à se poser la question de leur débogage. En effet, non seulement la mise au point séparée du code de chaque agent ne suffit pas à valider tous les aspects de l'application, mais la mise au point séparée elle-même ne peut se faire que si une simulation de l'environnement de l'agent est disponible.

Les systèmes multiagents impliquent, comme nous allons le voir, de nombreux processus qui tournent en parallèle sur une ou plusieurs machines. De ce fait, le débogage est rendu difficile car les bogs ne sont pas toujours simples à reproduire et à analyser et que, d'autre part, des « *trace*[1] » sur des processus isolés cassent la temporalité du système. Il importe donc de définir et de mettre en oeuvre des méthodes spécifiques pour le débogage de tels systèmes.

A cette fin, Spysma est un démon intégré à ALBA 1.0 qui permet de « *logger* » la plupart des évènements atomiques d'ALBA[2]. Phenix est un outil qui permet au programmeur d'exploiter ces « *logs* » afin de relancer individuellement un des agents de son SMA. Cette relance devant, bien évidemment, se réaliser dans les mêmes conditions que l'exécution du SMA que l'on désire analyser, des techniques permettent d'assurer la même temporalité. Le but de Phenix est donc de permettre d'isoler les problèmes, le programmeur pouvant ensuite mieux les analyser en utilisant les techniques classiques de débogage sur l'agent incriminé.

Nous avons pris le parti d'aborder ces outils bien qu'ils ne soient, a priori, pas directement liés au développement d'ALBA 1.0, et ce pour les raisons suivantes :

- Il est tout d'abord intéressant d'apporter un premier aperçu des difficultés qui peuvent naître de la programmation en environnement distribué, difficultés sur lesquels nous aurons bien évidemment l'occasion de revenir puisqu'elles ont constitué l'une des sources majeures des problèmes rencontrés durant le stage. Il est, dans tous les cas, souhaitable d'avoir une idée des moyens pouvant être mis en oeuvre pour y faire face.
- L'intégration de Spysma dans ALBA 1.0 constitue un autre point de critique. Il semble en effet peut souhaitable que des routines propres à Spysma soient disséminées de manière éparse dans le code d'ALBA. Nous renvoyons ici le lecteur à la partie consacrée à la bibliothèque d'évènements qui constitue la solution que nous avons envisagée pour ce problème.
- Ces outils devaient enfin être mentionnés, car bien évidemment, lorsque l'on développe une nouvelle version d'une bibliothèque existante il faut tenter d'assurer la compatibilité de cette dernière avec les outils qui sont bâtis dessus.

3.4 Description des concepts de ALBA 2.0

A mon arrivé au sein de Thales, j'ai eu à ma disposition le code source de la version 1.0 d'ALBA ainsi qu'un rapport partiellement finalisé décrivant les grands concepts d'ALBA et les évolutions à envisager pour la nouvelle version. Ma première tâche fut donc de comprendre les limitations d'ALBA 1.0 et d'étudier ce document. Ceci a permis de formaliser et rationaliser les grands concepts d'ALBA en un ensemble cohérent que nous tâchons de retranscrire ci-dessous. Nous allons donc décrire ici les grands principes qui régissent le fonctionnement d'ALBA et qui furent bien évidemment à la base des implémentations que nous décrirons ultérieurement. Si l'étude de l'existant nous a amené à nous intéresser à des cas particuliers, nous tâcherons ici d'être le plus général possible afin de respecter l'un des points clés de la philosophie d'ALBA à savoir la généricité maximale.

3.4.1 Définitions

> **ALBA :** c'est un ensemble de fonctionnalités permettant le développement et le déploiement d'un système multiagent

1 La commande « *trace* » permet le débogage des applications Prolog, il offre au programmeur la possibilité d'exécuter du code pas à pas et d'en contrôler l'exécution afin de l'analyser à son gré

2 Création d'agents, envoi de message, réception de message...

processus : c'est l'entité de même nom manipulée par le système d'exploitation sur lequel s'exécute ALBA (Windows, Unix, ...)

processus Prolog : c'est un processus créé par l'activation d'un exécutif Prolog. Les processus Prolog communiquent entre eux par sockets TCP/IP dont une implémentation est offerte par SICStus Prolog.

agent ALBA ou a-agent : c'est l'entité élémentaire que l'on peut produire en utilisant le niveau d'abstraction fourni par ALBA. C'est un agent au sens courant du mot mais son langage de commande est de bas niveau (échange de termes Prolog) et ne possède pas de sémantique. De même aucune structure d'agent particulière n'est imposée à un a-agent. Lorsque le contexte ne présentera pas d'ambiguïté nous utiliserons le terme « agent » pour « a-agent ». Un a-agent est identifié par un nom unique.

proto-agent : c'est l'ensemble des informations nécessaires à la création d'un a-agent (code Prolog, exécutable, mais également tous les fichiers de données nécessaires à l'a-agent...). Plusieurs a-agent peuvent être créés à partir du même proto-agent. A ce titre on peut rapprocher le concept de proto-agent de celui de classe des langages orientés objets, un a-agent étant une instance d'un proto-agent.

machine : caractérise le support physique sur lequel s'exécute un a-agent ou sur lequel réside un proto-agent. Une machine est identifié par son adresse IP.

SMA : l'ensemble des a-agents, des proto-agents et des machines sur lesquelles ils résident.

accointances : ensemble des a-agents connus d'un a-agent A donné. A peut communiquer par messages avec ses accointances, c'est ALBA qui se charge de toutes les opérations nécessaires aux communications, y compris, si cela est nécessaire, de la localisation et de la connexion avec l'a-agent à contacter.

Message : terme Prolog valide échangé entre deux a-agents.

3.4.2 Modèle libre des agents

L'état d'avancement actuel des recherches dans le domaine des systèmes multiagents n'a pas

encore permis de dégager un modèle d'agent - ou un ensemble de modèles - susceptible de satisfaire l'ensemble des applications potentielles. Il apparaît même assez clairement que les applications réalisées dans la division Aéronautique font appel à des modèles peu étudiés par la communauté scientifique (temps-réel, tolérance aux fautes…). Choisir un modèle a priori eût été une erreur risquant de faire aboutir à des conclusions erronées concernant l'intérêt industriel de la technologie multiagent. Alba doit permettre aussi bien le développement d'agents primitifs, c'est à dire pour lesquels aucune architecture particulière n'est imposée, que le développement d'architectures destinées à la mise en œuvre d'un modèle d'agent particulier. Cette flexibilité favorisera les expérimentations conduites par l'équipe. C'est ainsi que l'on pourra intégrer les résultats des travaux menés, tous orientés vers la définition d'architecture d'agents adaptés à des besoins spécifiques.

3.4.3 Proto-agents

Le proto-agent est l'ensemble des informations nécessaires à la création d'un a-agent (code prolog, exécutable mais aussi fichiers de données…). Plusieurs a-agents peuvent être crées à partir du même proto-agent. Le proto-agent sert à construire la représentation mémoire de l'a-agent ainsi que sa zone de données rémanentes.

Le proto-agent réside sur une machine particulière qui devra donc être connue de l'a-agent désirant créer un nouvel agent à partir du proto-agent. Ce dernier est donc caractérisé par :

- un nom de machine
- un répertoire sur cette machine contenant les données et instructions nécessaires à la création du processus Prolog ainsi que les données de travail de l'agent.

Les proto-agents eux-mêmes pourront être déplacés ou dupliqués d'une machine à l'autre, soit pour rendre la création d'agent plus efficace, soit pour pouvoir réaliser un code portable pour les a-agents créateurs.

3.4.4 A-agents

Un a-agent est un processus Prolog capable de communiquer avec d'autres a-agents (ses *accointances*) par une communication point à point.

Un a-agent est identifié par son nom – dont l'unicité est garantie par ALBA -.

Un a-agent s'exécute sur une machine – sa *machine courante*- et peut décider de se déplacer sur une autre machine qui deviendra la nouvelle machine courante de l'agent.

Chaque a-agent dispose sur sa machine courante d'une *zone de données rémanentes* sous la forme d'un répertoire qui sera déplacé lors de la migration de l'agent. Ce répertoire contient :

- les données nécessaires à la création du processus Prolog supportant l'a-agent.
- les données de travail rémanentes de l'a-agent.

Au démarrage (ou au re-démarrage) de l'agent, le répertoire courant sera le répertoire de la zone de données rémanentes. Ainsi, le programmeur n'aura pas à se préoccuper de la machine sur laquelle s'exécute l'a-agent.

Dans la suite du rapport, nous représenterons les a-agents comme indiqué sur le schéma ci-dessous. Un nom sera mentionné seulement lorsque cela est nécessaire.

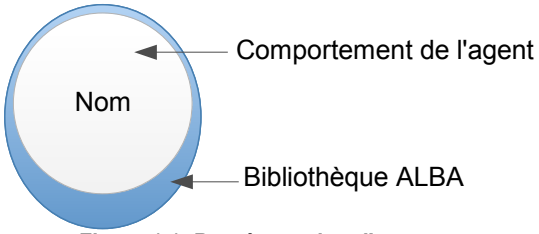


Figure 1.1: Représentation d'un a-agent

3.4.5 Agents étrangers ou Hétéro-agents

Il est possible d'inclure dans un SMA d'agents ALBA des agents n'ayant pas toutes les caractéristiques des a-agents mais obéissant à une convention minimale permettant de les traiter de manière uniforme.

Cette convention requiert que :

- l'agent étranger doit s'être connecté sur un port TCP/IP disponible
- l'a-agent devra pouvoir intégrer l'agent étranger en lui affectant un nom unique
- les agents échangeront des termes Prolog de la même manière que les a-agents entre eux

La plateforme NAOMI a été conçue pour être intégrée comme un agent étranger. D'une manière plus générale, l'agent étranger est le moyen proposé par ALBA pour intégrer dans un SMA n'importe quel programme susceptible de communiquer par TCP/IP.

3.4.6 Abstraction de la communication

3.4.6.1 Description

La gestion des communications est une tâche difficile nécessitant de nombreuses connaissances éloignées de la problématique des agents intelligents. Néanmoins, la communication est un élément essentiel de l'approche multiagent dans la mesure où elle représente l'unique moyen de partage entre les agents. C'est en effet en interdisant le recours à des partages d'informations complexes (données communes, sémaphores…) que les systèmes multiagent peuvent prétendre réduire la complexité structurelle des systèmes réalisés.

La communication est du type point-à-point. L'agent destinataire, tout comme l'expéditeur, est identifié par son nom. Les mécanismes permettant d'identifier les agents et de transmettre effectivement l'information sont entièrement à la charge d'Alba.

Alba doit donc offrir des possibilités d'échanges entre agents aussi puissantes que possible mais décharger le programmeur de la gestion de ces échanges. En conséquence, Alba devra permettre à chaque paire d'agents d'échanger des messages constitués de termes Prolog valides (finis ou infinis). Ainsi, du point de vue du programmeur d'un agent A, tout agent B connu de A est susceptible d'être destinataire de messages de A ou expéditeur de messages vers A.

Les messages successifs sont indépendants les uns des autres et donc lorsqu'ils contiennent des variables, les liens de variables entre deux messages ne sont pas conservés. Si ce besoin se faisait sentir, il sera néanmoins facile pour le programmeur de mettre en place un protocole permettant de restaurer ces liens (par exemple en adjoignant à chaque message proprement dit, la liste des variables du problème).

3.4.6.2 Ordre de lecture des messages

Le programmeur ALBA pourra s'appuyer sur l'hypothèse suivante concernant l'ordonnancement des messages.

```
Pour un même couple de a-agents, l'ordre des messages à l'émission est le
même à la  réception, soit plus formellement: si m1 et m2 sont deux messages
émis par A vers B tels que m1 est émis avant m2, B recevra m1 avant m2.
```

Cette hypothèse pourra être utilisée par l'implémentation d'Alba (par exemple pour la gestion de la migration) ainsi que par le programmeur implémentant un protocole particulier.

```
En revanche si A envoie un message m1 vers B, C envoie un message m2 vers B,
avec m2 envoyé postérieurement à m1, rien ne garantit que m2 sera lu après
m1. Ce n'est pas un problème en pratique.
```

3.4.6.3 Gestion des erreurs

L'abstraction de la communication réalisée concerne également la gestion des erreurs. Ce point est très important car d'une part, il ne faut pas surcharger le programmeur de considérations de bas niveau complexes à gérer et d'autre part, il faut assurer le fonctionnement correct (et en particulier sans blocages ni plantages) du SMA. Pour cela, aucune erreur de communication ne remontera vers l'a-agent. C'est Alba qui se chargera de la gestion proprement dite de la communication (connexion, transmission, déconnexion propre...) et du traitement des erreurs[1]. Du point de vue de l'agent les principes suivants sont à retenir :

- un message émis peut ne pas arriver sans que l'expéditeur en ait conscience autrement que par les effets de sa non transmission (absence de la réponse conventionnelle au message dans un temps donné). Si l'acheminement s'avère impossible (connexion impossible, communications coupées, perte du message…) les agents expéditeurs ne seront donc pas informés de l'échec et le message sera perdu. La métaphore à considérer ici est celle du courrier postal ordinaire lorsque l'expéditeur n'a pas marqué son adresse au dos de l'enveloppe. Il appartient donc aux agents, dans les cas où cela s'avère nécessaire, de se protéger vis à vis de la perte possible d'un message grâce à des protocoles spécifiques à mettre en place.
- la notion de connexion et donc d'atteignabilité d'une accointance est gérée par Alba et n'est pas remontée au niveau de l'agent. Ce dernier ne peut que se reposer sur un protocole de plus haut niveau (accusé de réception) pour savoir si la liaison est possible. Si ce n'est pas le cas, il n'a pas d'autres solutions que d'attendre puis de re-essayer ou de « changer d'accointance », tout ayant été fait par ALBA pour tenter de la joindre.

Il faut noter également que même si il a bien reçu un message, un agent peut ne pas y répondre, soit parce qu'il est occupé à autre chose soit qu'il décide de ne pas y répondre (autonomie des agents). Cela oblige donc les agents à traiter l'absence de réponse (un « *timeout* » est prévu à cet effet) traitement qui englobera les problèmes liés à l'échec d'un envoi de message.

3.4.7 Machine

Les SMA construits peuvent être déployés sur plusieurs machines et les agents peuvent migrer librement (c'est à dire de leur propre initiative) d'une machine à l'autre.

Une machine est identifiée par son nom, correspondant à son adresse réseau ou son adresse IP elle-même. Ce nom n'est utile au programmeur que pour la création dynamique d'agent et pour la migration (cela va de soi).

Afin de permettre la programmation des agents indépendamment de la répartition des agents sur les machines, et donc de réaliser des SMA portables, la machine '*localhost*' désignera la machine sur laquelle s'exécute un agent à un instant donné. Il sera ainsi possible, dans les cas simples, de ne pas se soucier de la distribution sur les machines, sans pour autant renoncer à cette distribution, par exemple par migration des agents sur des machines dont le nom est obtenu dynamiquement.

Un a-agent aura accès :

- à sa zone de données rémanentes. Seule cette zone migre avec l'agent,
- à tous les fichiers de la machine '*localhost*' à l'aide d'une adresse relative à cette machine

3.4.8 Multimachine et migration

3.4.8.1 Description

Les agents Alba doivent pouvoir être créés sur une machine différente de celle sur laquelle se situe l'agent initiateur de la création. Ils doivent pouvoir également librement migrer d'une machine à l'autre.

Cette exigence a plusieurs conséquences :

1 Cette position, outre le fait que le principe semble bien adapté aux objectifs définis en terme d'agents autonomes, est rendue nécessaire par la confusion régnant dans la gestion des anomalies autour de la communication. Il est clair que de tels détails ne doivent pas remonter au programmeur d'agents au risque de le noyer dans la complexité de traitements dont il ne doit pas avoir à se préoccuper au niveau d'abstraction auquel on souhaite le placer.

- Les agents doivent pouvoir accéder aux ressources de la machine qui les héberge (par exemple, création de fichiers de travail temporaire),
- Les agents doivent pouvoir également accéder aux ressources des machines dont ils sont originaires (bases de données trop volumineuses à transporter)

L'architecture de programmation retenue doit permettre la création d'un agent sur n'importe quelle machine, et a fortiori sa migration, sans aucune modification de code de l'agent lui-même.

La création d'agent pose un problème particulier dans la mesure où il est techniquement nécessaire de disposer du code de l'agent à créer (proto-agent) sur la machine hôte. La création d'agent étant avant tout dynamique (tout agent peut en créer un autre à tout moment), être invariant dans une migration nécessite des moyens de rapatriement du code du proto-agent sur la machine sur laquelle s'effectue la création.

3.4.8.2 Intérêts

La migration présente divers intérêts. On peut tout d'abord lui reconnaître un intérêt théorique dans le sens où si les agents sont capables de migrer dynamiquement d'une machine à l'autre, parfois même à plusieurs en même temps, tout en poursuivant leurs tâches et en continuant à communiquer de manière transparente, cela permet de valider la conception de notre modèle.

La migration revêt également des intérêts pratiques notables :

- Déploiement multimachine dynamique : il s'agit de pouvoir s'adapter très vite à une configuration réseau existante, par exemple lors d'une démonstration d'applications où la configuration disponible peut n'être connue qu'au dernier moment. Il doit ainsi être possible, par exemple, de charger l'ensemble d'un SMA sur une machine donnée puis de faire migrer certains agents sur les autres machines accessibles. Une telle solution est préférable à une approche statique nécessitant la création préalable de tables de configuration associant agents et machines hôtes, car elle permet de s'adapter à la configuration locale sans avoir à modifier de fichiers au préalabre.
- Répartition des charges de calcul et de la mémoire utilisée, l'idéal serait bien sûr de pouvoir déployer un agent par machine.

3.4.9 Démon ALBA

Le fonctionnement en multi-machine d'un SMA réalisé avec ALBA nécessite de pouvoir exécuter des commandes à distance (comme par exemple la création d'un processus Prolog et son activation). Afin d'être le plus possible indépendant du système d'exploitation d'une part et de ne pas être contraint par les règles de sécurité propre au réseau sur lequel le SMA s'exécute, toutes les commandes nécessaires seront implémentées par un démon à activer sur chacune des machines connues du SMA. Ce démon utilisera un ensemble minimum de ressource réseau (TCP/IP, # de port fixe). Les commandes implémentées par ce démon permettront :

- le transfert de fichier d'une machine à l'autre
- le lancement d'un processus Prolog

3.4.10 Comportement d'un agent

Il est décrit par deux prédicats à implémenter par le programmeur dans le corps du proto-agent. Ces prédicats sont *behavior* qui décrit le comportement de l'agent lors de sa création initiale et *restart* qui décrit le comportement de l'agent à l'issue de chaque migration. Ces deux prédicats sont appelés automatiquement par ALBA. Ils seront approfondis dans la partie consacrée à l'implémentation.

3.4.11 Priorité à la simplicité de programmation

Ce doit être l'un des objectifs important des choix guidant la conception d'ALBA que de faciliter la programmation des agents. Il est en effet nécessaire dans le contexte de prototypage qui est celui de l'équipe de pouvoir se concentrer sur la recherche de solutions au problème posé plutôt que sur les détails

techniques de leur mise an œuvre.

Cet objectif doit être pris en compte même au prix d'une perte de généralité des solutions retenues.

3.4.12 Bilan synthétique des axes directeurs

Avant de s'engager dans un discours plus technique qui fera apparaître le travail effectué sur la nouvelle version d'ALBA, il nous est apparu souhaitable de récapituler ici les axes majeurs qui ont dirigé le développement et qu'il a été nécessaire de conserver à l'esprit durant tout le stage :

- rester le plus générique possible, aucune restriction ne doit être faite sur les agents du système
- décentraliser un maximum les données
- penser en terme d'autonomie des agents et d'interdépendance limitée
- donner la priorité à la simplicité de la programmation
- gérer les erreurs de manière transparente pour l'utilisateur qui doit pouvoir se concentrer sur des problèmes de plus haut niveau
- toujours réfléchir en terme de déploiement multimachine
- se concentrer sur la portabilité (Windows, Unix) et l'efficacité du code notamment en ce qui concerne les routines de communication destinées à être appelées intensivement et nécessitant donc d'être optimisées

3.4.13 Schéma d'ensemble

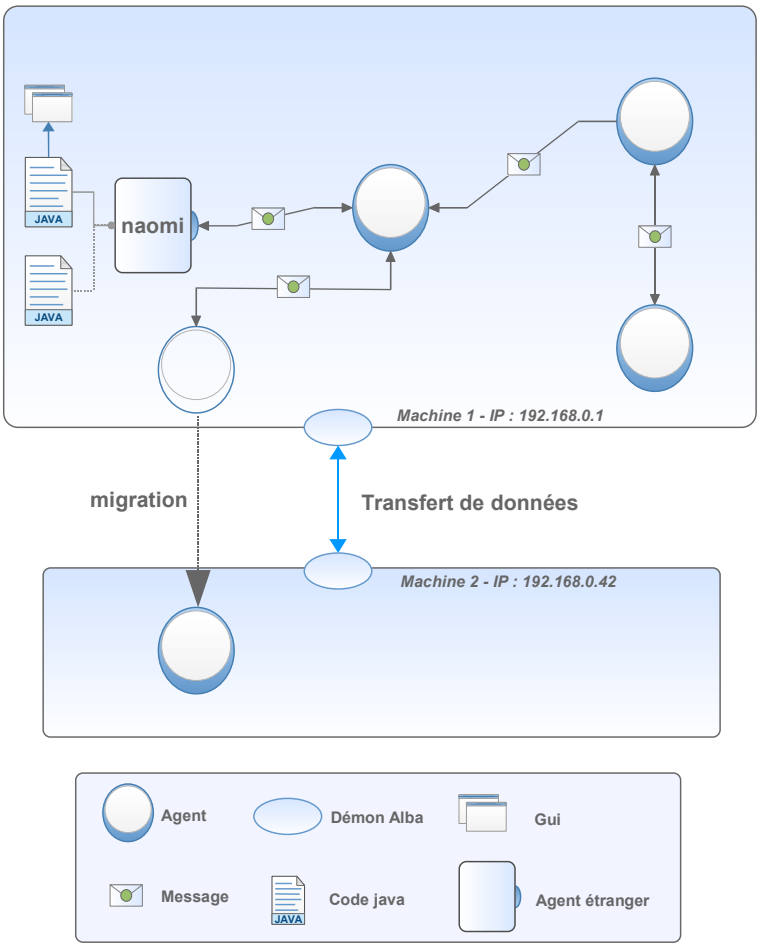


Figure 1.2: Schéma d'ensemble

3.5 Travail réalisé sur ALBA 2.0

Nous allons nous concentrer ici sur l'implémentation à proprement parler d'ALBA 2.0. Nous traiterons les différents points par thème en abordant à chaque fois les problèmes qui se sont posés, les alternatives proposées, les solutions adoptées et enfin les améliorations envisageables. Cette approche offrira une bonne vue d'ensemble du fonctionnement interne d'ALBA et nous permettra d'étudier les fonctionnalités qu'elle offre aux programmeurs d'agents.

3.5.1 Gestion de la mémoire

Comme nous l'avons déjà mentionné, l'une des différences fondamentales entre ALBA 1.0 et ALBA 2.0 réside dans la représentation mémoire des données internes de la bibliothèque. Là où la précédente version d'ALBA utilisait des variables « *mutables* » qui étaient passées à chaque prédicat, nous avons choisi de stocker toutes les données en mémoire par le biais des prédicats *assert* et *retract* de Prolog. Il devient alors inutile de passer une variable à tous les prédicats puisque ces données sont accessibles de n'importe quel endroit du programme par simple unification sur la mémoire de l'agent.

Assert

```
asserta(P) : permet d'ajouter P au début de la base de données

assertz(P) : permet d'ajouter P à la fin de la base de données
```

Retract

```
retract(P) : permet de retirer la première clause s'unifiant avec P de la
base de données

retractall(H) : permet de retirer toutes les clauses dont la tête s'unifie
avec H
```

Cette gestion de la mémoire présente donc plusieurs avantages :

- les données sont accessibles de partout sans passer de variable explicite et ce de manière très efficace grâce aux tables de hachage internes de SICStus Prolog. Le fait de ne pas avoir à passer de variable interne à tous les prédicats n'est pas un réel avantage en soi mais il permet tout de même de ne pas perturber l'utilisateur de la bibliothèque avec des données internes qui n'ont pas de sens pour lui
- les informations sont persistentes au « *backtrack* » ce qui nous permet de nous affranchir de certaines difficultés ayant nuit au fonctionnement de la version 1.0
- il est possible de faire des traitements avancés sur ces données en utilisant les prédicats Prolog d'unification sur les faits en mémoire.

Les appels à *assert* et *retract* ont évidemment été encapsulés dans un prédicat supérieur ce qui est plus propre, et qui pourrait nous permettre de changer de politique de gestion mémoire avec des conséquences réduites. Cette solution nous permet d'autre part d'effectuer simplement des traitements automatiques à chaque fois que l'on fait un effet de bord dans la mémoire ce qui peut s'avérer utile comme nous le verrons ultérieurement. (Cf. Bibliothèque d'évènements)

Notons toutefois que ce genre d'utilisation de Prolog doit rester marginale, l'effet de bord ne s'inscrivant pas dans la philosophie de la programmation logique qui est basée sur la logique du premier ordre, sur la récursivité et l'unification. Rappelons à cet égard que l'un des arguments justifiant le recours aux effets de bord est que la communication elle-même est un effet de bord, il paraît donc difficile d'envisager de développer un outil reposant sur la communication sans en utiliser.

Au fur et à mesure de l'avancement de cette section, nous verrons quelles sont les données internes utilisées et comment elles sont représentées. Nous dresserons une vue globale de cette mémoire interne avant d'aborder la migration des agents qui en nécessite une analyse rigoureuse.

3.5.2 Identification des a-agents

Comme nous l'avons mentionné dans les concepts fondateurs de la bibliothèque, ALBA 2.0 doit assurer l'unicité du nom des agents. Alba 1.0 identifiait les agents grâce à une « *IDCARD* » qui contenait en sus du nom de l'agent, son adresse et son numéro de port (Cf. Démon Alba) et c'est par celle-ci que le programmeur d'agent interagissait avec ses accointances.

ALBA 2.0 ayant pour vocation de masquer toutes ces informations internes à l'utilisateur, il a été décidé d'identifier les agents par un simple nom qui sert ensuite dans tous les prédicats de la bibliothèque pour communiquer avec cet agent. De ce fait, il est fondamental d'assurer l'unicité du nom de chaque agent dans le système. Il eût été envisageable de reléguer cette tâche au programmeur, mais cette solution par trop restrictive n'a évidemment pas été jugée satisfaisante.

L'unicité des noms est donc gérée automatiquement comme suit. Les noms sont des termes Prolog qui ont la forme suivante : *NomDuPère* / *NomAgent.* Le premier agent du système n'ayant pas de père il garde son nom original, par exemple *root*. Si *root* crée un agent nommé *toto*, le nom complet du nouvel agent est donc *root/toto*. Cela ne répond que partiellement au problème, car l'agent *root* peut créer plusieurs agents du nom de *toto.* Dans ce cas, les noms complets des agents créés seraient : *root/toto*, *root/toto(1)*, *root/toto(2)*... Nous voyons donc comment par ce procédé on assure l'unicité des noms en ayant uniquement à sauvegarder des informations locales à l'agent. Nous restons donc dans notre optique de distribution maximale des données. Notons également que cette représentation sous forme de terme permet très rapidement de faire des traitements sur le nom pour obtenir par exemple les noms de tous les ancêtres d'un agent.

3.5.3 Gestion des « *timeout* »

De nombreux prédicats d'ALBA pouvant être, potentiellement, des appels bloquants, présentent un argument de type « *timeout* ». Il n'est en effet jamais souhaitable de se bloquer et d'attendre indéfiniment une information. Le format interne utilisé par SICStus Prolog pour les « *timeout* » est le suivant *sec*:*usec*. Ce format est peu intuitif, il n'est pas satisfaisant et peut conduire à des erreurs. Ainsi 5:2 repésente 5,000002 secondes et non 5,2 secondes, pire 5:9000000 représente 14 secondes.

Nous avons donc pris le parti pour ALBA 2.0 d'utiliser le format suivant, des conversions transparentes pour l'utilisateur étant faite par la bibliothèque vers le format interne de SICStus :

- *off* : temps illimité, peu recommandé en pratique
- entier : nombre entier de secondes à attendre
- flottant : nombre flottant de secondes à attendre par exemple 5.25 secondes.

3.5.4 Le démon ALBA

Le démon ALBA doit être lancé sur chaque machine intervenant dans le SMA. Son fonctionnement est très simple. Avant de le décrire plus en avant rappelons très brièvement quelques éléments de programmation réseau qui seront utilisés par la suite.

Quelques élements de programmation réseau

Adresse IP : il s'agit d'une adresse numérique permettant d'identifier une machine sur le réseau.

Numéro de port : une machine comprend généralement 65536 numéros de ports dont les premières plages sont réservées. Ce numéro permet d'identifier vers quel processus il faut acheminer les paquets d'informations circulant dans le réseau.

Socket : il faut voir les sockets comme des tubes bidirectionnels qui permettent de lire et d'écrire des données en locale ou bien à distance.

```
Stream : il s'agit d'une surcouche des sockets. Les streams correspondent
aux pointeurs de fichiers utilisés au niveau du système d'exploitation. Les
streams peuvent ensuite être passées à différentes fonctions d'entrée/sortie
afin de lire ou d'écrire dessus.
```

Lorsqu'on exécute le démon ALBA sur une machine, il ouvre un stream sur un port donné[1] et se met en écoute dessus. Le démon peut recevoir deux types de requête de la part des agents du système, soit une requête de migration, soit une requête de création à distance. Dans les deux cas, il permet le rapatriement des données utiles à la création du nouvel agent et se charge d'exécuter le nouveau processus de l'agent en local.

Il paraît guère utile de s'étendre sur la question. Signalons toutefois que, compte tenu du fonctionnement interne de Prolog les transferts sont très lents. Il est nécessaire, en effet, de transférer les données octet par octet afin de s'assurer de la portabilité du système sur différentes architectures et de la cohérence des informations transmises. Les streams étant une encapsulation Prolog de routines de plus bas niveau, il est en pratique relativement lourd de faire les conversions, Prolog – représentation interne, représentation interne – Prolog, pour chaque octet transféré. Nous voyons là, un des rares cas, où le langage Prolog montre ses limites. Il serait donc souhaitable à l'avenir de programmer les transferts de données en C et d'interfacer ce code avec le code Prolog du démon afin de pallier ce problème. Ceci est tout à fait envisageable compte tenu des facilités d'interfaçages avec le C qu'offre l'implémentation de SICStus.

3.5.5 Création des agents

La création des agents est une opération évidemment nécessaire à toute plate-forme. Nous allons détailler ici les différents cas qui se présentent.

3.5.5.1 Initialisation du système

Si les agents peuvent créer d'autres agents, il est bien évidemment indispensable de pouvoir offrir à l'utilisateur un point d'entrée permettant de lancer le premier agent du système. Cela se fait dans ALBA par l'intermédiaire d'un script de lancement que l'on appelle comme suit :

Lancement du premier agent

```
call agentLauncher.bat comportement -a nom -c contacts -p parameter -w
window
```

Bien évidemment le script agentLauncher.bat doit être adapté pour tourner sur toutes les plate-formes supportées.

Ce script appelle en pratique la même routine que celle appelée lorsqu'un agent crée un autre agent en local. Nous allons donc détailler les paramètres du script dans la section suivante.

3.5.5.2 Création locale

La création d'un agent depuis le code d'un autre agent se fait par l'appel suivant :

```
create_agent(+Host, +Name, ?Parameter, +ContactsName, +Behavior, +Window,
-FullName)

Host : adresse de la machine sur laquelle doit être créé l'agent
Name : nom que l'on souhaite donner à l'agent
ContactsName : liste des accointances fournies à l'agent
Behavior : chemin vers le comportement de l'agent
Window : with ou without selon que l'on souhaite avoir une fenêtre
apparente   pour l'agent
FullName : nom complet qui identifie l'agent dans le système
```

1 Le même sur chaque machine

La création se fait en local lorsque le nom d'hôte fourni est *localhost*. Dans ce cas, c'est l'agent lui même qui génèrera la commande adéquate à la création de l'agent et qui l'exécutera. En pratique il s'agit d'un appel à l'exécutable de SICStus avec les paramètres qui conviennent.

Les accointances sont fournies sous la forme d'une liste de contacts : [Contact1, ..., ContactN]. Chaque contact peut prendre deux formes :

- Nom complet d'un agent
- Nom:Host:Port

ALBA accepte les fichiers de comportement suivant :

- .pl, .ql, .po qui sont exécuté par un appel à SICStus
- .exe exécuté directement
- archive .rar, le fichier est alors décompressé, il contient un *manifest* qui indique le fichier de comportement. L'archive peut contenir des fichiers et données utiles à l'agent

FullName est renvoyé par le prédicat, il a la forme décrite dans Identification des a-agents. C'est ce nom complet qui devra être utilisé ensuite par le programmeur pour intéragir avec l'agent nouvellement créé.

3.5.5.3 Création distante

La création distante se fait lorsqu'on utilise *create_agent* avec une adresse d'hôte distante. La seule différence avec la création locale est la suivante : au lieu que ce soit l'agent qui lance directement un nouveau processus Prolog, dans ce cas l'agent communique directement avec le démon ALBA de l'hôte visée, il transfert les fichiers de comportement de l'agent à créer et c'est le démon qui exécute le nouveau processus Prolog.

3.5.5.4 Initialisation de l'agent

Dans tous les cas l'agent est initialisé de la même manière. Il ouvre une *stream* sur le premier port disponible de la machine où il réside. Cette *stream* permettra aux autres agents d'entrer en contact avec lui comme cela est décrit dans Initalisation de la communication. Il initialise alors ses données internes et génère son « *IDCARD* » qui est un terme de la forme *idcard(FullName, Host, Port)*. Il cherche ensuite à se connecter avec ses accointances dont notamment et de manière systématique son père même si ce dernier n'est pas passé explicitement dans ses accointances. La tentative de connexion peut se faire lorsque l'agent dispose des informations d'hôte et de port du contact à joindre. C'est en pratique le cas lorsque l'on utilise la notation Nom:Host:Port. Si le programmeur se contente de donner un nom deux cas de figure sont alors envisageables :

- soit l'agent créateur passe le nom d'un contact avec qui il est lui même connecté. Dans ce cas l'agent créateur passe à l'agent créé et de manière transparente pour le programmeur les informations de connexion. La connexion est alors établie automatiquement,
- soit l'agent créateur ne dispose que d'un nom dans ce cas aucune connexion effective n'est établie

Dans tous les cas il faut avoir à l'esprit que ces informations peuvent être erronées ou périmées auquel cas la connexion ne pourra être établie. Nous verrons comment ces considérations ont fortement influé sur les routines de communication.

Une fois l'agent initialisé, ALBA appelle le prédicat suivant qui doit être défini dans le code de l'agent à lancer et qui décrit son comportement :

```
behavior(+Name, ?Parameter, +ContactsName)

Name : nom complet de l'agent créé
Parameter : paramètre facultatif de lancement
ContactsName : accointances de l'agent
```

Le père de l'agent joue un rôle particulier, puisqu'à l'initialisation c'est le premier agent avec lequel se

connecte l'agent nouvellement créé, les informations permettant de contacter le père sont d'ailleurs toujours passé à son fils. Nous avions d'abord pris le parti de passer également le nom du père dans les paramètres de behavior mais, comme nous l'avons dit, il est extrêmement simple d'obtenir le nom du père à partir du nom complet d'un agent. Le prédicat suivant permet donc à un agent d'obtenir le nom de son père. Par convention le père du premier agent du système a pour nom *__alba__*.

```
get_father_name(?FatherName)

FatherName : nom complet du père de l'agent appelant
```

3.5.5.5 Attente de création

Un agent créant un autre agent peut avoir l'impératif besoin d'être certain que la création a réussi et qu'il peut communiquer avec son fils. Le processus de création pouvant être long notamment lors de création distante d'agent ayant des fichiers de données volumineux, il est possible d'utiliser le prédicat suivant :

```
wait_for_creation(+Name, +Timeout, -Status)

Name : nom complet de l'agent à attendre
Timeout : délai maximal d'attente
Status : true si le fils s'est connecté dans les délais, false sinon
```

3.5.6 Communications

Les communications entre agents sont la clé de voûte du système. Il est donc fondamental de se concentrer sur cette question. Un soin particulier a bien entendu était pris pour optimiser ces routines.

3.5.6.1 Initialisation de la communication

Avant de pouvoir s'envoyer des messages une connexion doit être établie entre deux agents. Pour ce faire, un agent A disposant de l'adresse et du port d'un agent B crée une *stream* et utilise ces informations pour la connecter à l'agent B, qui comme nous l'avons dit est toujours à l'écoute sur ce port. A envoie alors à B les informations nécessaires à la connexion. A et B stockent alors en interne les informations relatives à leur contact respectif. Nous détaillerons plus loin comment cela est représenté en mémoire. Une fois la connexion établie les agents peuvent évidemment échanger des messages sans réitérer cette procédure.

Il est également possible d'ajouter à la main, à tout moment, un agent étranger au système à l'aide du prédicat suivant :

```
add_foreign_contact(+Name, +Host, +Port)

Name : nom donné à l'agent étranger
Host : adresse de l'agent étranger
Port : port de l'agent étranger
```

Cela peut, par exemple, permettre de lancer deux systèmes multiagents n'ayant aucun lien, a priori, entre eux puis de les lier à tout moment. Ils ne forment alors qu'un seul réseau qui peut échanger des informations par messages comme si ils avaient toujours formé le même ensemble. Notons à ce propos que l'unicité du nom donné à l'agent étranger doit être assuré par le programmeur. Il est d'autre part souhaitable que les agents initiaux de chaque système porte un nom différent ce qui permet d'éviter d'avoir des redondances de nom dans le système unique émergent ce qui entrainerait des comportements indésirables.

3.5.6.2 De l'usage de deux canaux de communication

L'implémentation SICStus ne permet pas de mettre en place des *threads*. Il aurait pourtant été

souhaitable d'avoir un *thread* s'occupant des tâches internes d'ALBA et de relever les nouveaux messages à leur arrivée, tandis qu'un autre *thread*, pouvant être activé à tout moment par le premier, se serait chargé du comportement de l'agent. Comme nous allons le voir, il existe un certain nombre de messages internes à ALBA dont le traitement est utile pour le bon fonctionnement du système (Cf. Messages de service). Dans ALBA 1.0, tous les messages, que ce soit ceux échangés par les agents ou les messages interne, circulaient par une seule *stream*, ce fonctionnement ralentissait le traitement des messages internes et entrainaît des décalages temporels dans la lecture des messages. Nous avons donc pris le parti de séparer les canaux de circulation de messages « classiques » et de messages internes. Ainsi lors de la première connexion entre deux agents, une paire de canaux est créé entre ces derniers comme indiquait sur le schéma suivant :

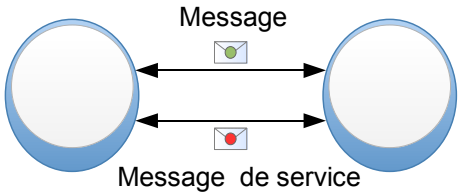


Figure 1.3: Deux canaux de communication

Dès lors, à chaque appel d'une routine d'ALBA, la routine *core_treatments* est appelé. Cette routine se charge de lire tous les messages internes d'ALBA, ci-après nommés messages de service, et de les traiter. Il est donc essentiel qu'un agent ALBA fasse fréquemment appel aux routines de la bibliothèque afin de présenter un comportement satisfaisant pour le système. Cette hypothèse n'est, somme toute, pas aberrante puisque le propre d'un agent désirant intéragir avec ses pairs est bien évidemment de consulter ses messages. Ceci ne pose d'ailleurs pas de réel problème en pratique car les automates ou les fils de raisonnement utilisés pour décrire le comportement des agents font automatiquement appel aux routines d'ALBA à pas de temps régulier.

Une difficulté se pose lorsqu'un agent se lance dans un calcul très long ou pire part en boucle infini suite à une erreur de programmation, dans ce cas évidemment l'agent ne relève plus ses messages et ne peut plus intéragir avec le système. Ce problème peut-être résolu ainsi : si A sait qu'il a un calcul potentiellement coûteux à effectuer, il crée un agent B pour que celui-ci effectue le calcul pour lui, une fois le calcul fini, B envoi le résultat à A et disparaît, pendant ce temps A a tout le loisir d'intéragir avec le système. D'autres solutions plus avancées ont également été étudiées par François Chazal durant son stage.

Il n'est pas exclu qu'un jour ALBA utilise des *threads* différents si une version prochaine de SICStus le permet, nous ne sommes toutefois pas persuadé que cette complexité supplémentaire serait réellement bénéfique au système.

3.5.6.3 Envoi de messages

ALBA 2.0 propose deux prédicats pour l'envoi des messages, le premier permet un envoi simple de message et doit être utilisé en général, le second permet d'envoyer un message et d'attendre un accusé de réception pendant un temps donné. Il est préférable d'utiliser cette routine avec parcimonie i.e. lorsque le contexte l'exige expressément.

Messages sans acquittement

```
send_message(+Message, +Recipients)

Message : message à envoyer
Recipients : nom des destinataire du message, il peut s'agir d'un nom
simple ou d'une liste de noms
```

Voici l'algorithme qui est utilisé pour l'envoi de message :

Algorithme d'envoi de message

```
1. Si l'agent s'envoie un message à lui-même
   alors placer le message dans la boîte au lettre

2. Si nous sommes connectés avec le destinataire
   alors envoyer le message dans la stream de message qui lui est associée
     Si l'envoi échoue
     alors nettoyer les informations du contact
           rappeler la routine d'envoi de message

3. Si nous n'avons pas d'informations sur le contact et que le contact n'est
pas un agent que nous venons de créer
   alors rechercher le contact
         mettre le message dans la file des messages en attente
   sinon mettre le message dans la file des messages en attente
```

Il est possible d'envoyer un message particulier : ping(ID). A la réception d'un tel message, le message pong(ID) sera automatiquement renvoyé à l'expéditeur par ALBA.

Messages avec acquittement

```
send_message_and_wait_till_received(+Message, +Recipient, +Timeout,
-Status)

Message : message à envoyer
Recipient : nom du destinataire
Timeout : temps maximal d'attente de l'accusé de réception
Status : true si l'accusé de reception a été reçu false sinon
```

Afin d'assurer le bon fonctionnement de cette routine, il a été nécessaire d'associer chaque message envoyé à un identifiant numérique qui s'incrémente à chaque message. En pratique A envoie un message à B avec l'identifiant correspondant, il se met alors en attente pendant au plus *Timeout* secondes. Pendant ce temps, il exécute ses tâches élémentaires dont principalement le traitement des messages de service. Lorsque B reçoit le message, la couche interne d'ALBA envoie automatiquement l'accusé de réception à A sur sa *stream* de service avec l'identifiant original. Si A reçoit le message dans les délais, il sort de son attente et reprend son travail.

De l'usage des files de messages en attente

La bibliothèque tentant de faire le maximum pour assurer que les messages sont acheminés avec succès, il a été décidé de placer les messages ne pouvant pas être envoyés immédiatement dans une file d'attente. Il existe une file d'attente par nom. Lorsqu'un agent se connecte à nous, suite à une recherche par exemple, les messages en file d'attente qui lui étaient destinés lui sont immédiatement envoyés. Il faut noter que ce procédé trouve également son utilité lors de la création d'un agent. Lorsque A crée B, il ne connait pas les informations lui permettant de joindre B (le port de B notamment, puisque celui-ci se verra attribué le premier port disponible de la machine sur laquelle il est créé). Si A veut envoyer un message à B tout de suite après avoir lancé sa création et sans faire d'appel à la routine *wait_for_creation* un problème se pose. Or nous ne voulons pas forcer A à attendre la création de B, ce qui est une source potentielle de blocage momentané du système et ralentit l'exécution de A. Avec les files d'attente, tout se passe de manière transparente. A envoi son message à B, ne pouvant l'envoyer effectivement, il met le message dans la file des messages en attente pour B. Lorsque B est créé, celui-ci se connecte à son père qui lui envoie automatiquement les messages qui étaient en attente.

Bien évidemment, afin de ne pas surcharger la mémoire et le SMA, des précautions ont été prises. C'est ainsi que les files sont des files circulaires dont la taille est limitée par une variable du système. Nous avons utilisé une représentation particulière des listes qui nous permet de concaténer deux listes en temps constant.

Ce mode de fonctionnement soulève toutefois des objections. Est-il vraiment souhaitable qu'un agent reçoive une masse de messages d'un coup dont certains potentiellement très anciens. Nous pensons que

l'agent doit être capable de lui-même de gérer les messages qu'il reçoit, ALBA effectuant le maximum de traitement pour qu'ils arrivent tous à leur destinataire.

3.5.6.4 Réception de messages

La réception des messages est basé sur le prédicat *socket_select* offert par SICStus, il convient donc d'en étudier sommairement le fonctionnement :

socket_select

```
socket_select(+TermsSockets, -NewTermsStreams, +TimeOut, +Streams,
-ReadStreams)

TermsSockets : gestion des nouvelles connexions sur la liste des sockets
fournis
NewTermsStreams : renvoi une liste de stream ouvertes suites aux nouvelles
connexions
TimeOut : timeout
Streams : liste de streams à écouter
ReadStreams : renvoi de la liste des streams ayant des données à lire
```

Bien que nous ne disposions pas de *threads*, les appels au prédicat *socket_select* nous permettent de disposer momentanément des avantages de ces derniers. Il est donc fondamental d'en tirer le meilleur parti.

Grâce à des appels successifs à *socket_select* avec un *timeout* de 0 et une seule *stream* X à écouter, il est possible de récupérer tous les messages en attente dans X. C'est une possibilité qui avait été négligée dans ALBA 1.0, mais qui peut s'avérer très utile.

Un message

```
read_message(-Message, ?Sender, +Timeout)

Message : message lu
Sender : expéditeur du message. Il est possible d'instancié Sender pour ne
lire que les messages émanant d'une accointance particulière
Timeout : temps maximal d'attente de message
```

Algorithme de lecture de message

```
1. Si la boîte aux lettres n'est pas vide
   alors retirer le premier message de la boîte

2. Si le temps n'est pas écoulé
   alors écouter les streams de lecture de toutes les accointances
        lire un message par stream contenant un message et le placer dans
        la boîte aux lettres
        mettre à jour le timeout
        rappel récursif

3. Si le temps est écoulé sans nouveau message
   alors renvoyer timeout comme message et __alba__ comme expéditeur
```

En pratique les traitements sont un peu plus complexes que cela. Un soin tout particulier doit être porté à la mise à jour des *timeout*. Il est d'autre part nécessaire d'écouter, en plus de toutes les *streams* de message des accointances, toutes leurs *streams* de service. Il a donc été nécessaire de mettre en place un dispositif permettant de séparer une liste de *streams* hétérogènes en une liste de *streams* de message et une liste de *streams* de service. Ceci permet à un agent de continuer à effectuer les tâches internes à ALBA lorsqu'il attend un message. Ainsi même si un agent se met en lecture de message avec un *timeout* à *off,* et qu'il ne reçoit aucun message pour le débloquer, il ne sera pas un agent « mort » pour le SMA.

Afin de se prémunir contre un agent qui enverrait une quantité inconsidérée de messages[1] et de porter à chaque accointance une même attention, nous avons pris le parti de lire un message par *stream* à la fois.

Notons que les conditions imposées sur l'ordre des messages (Cf. Ordre de lecture des messages) sont respectées. Ceci nous est assuré par le protocole TCP/IP qui acquitte chaque packet et par le fonctionnement des *streams*.

Tous les messages

Il peut être nécessaire d'établir des stratégies de lecture des messages. Dans cette optique, il est intéressant de pouvoir récupérer l'ensemble des messages existants à un moment donné. C'est ce que permet le prédicat :

```
read_all_messages(-Messages, ?Senders)

Message : liste de listes de messages lus
Senders : liste d'expéditeurs correspondant à chaque liste de messages. Il
est possible d'instancié Senders pour ne lire que les messages de
certaines accointances d'intérêt.
```

Algorithme de lecture de tous les messages

1. Lire tous les messages de toutes les *streams* à considérer et les placer dans la boîte aux lettres
2. Lire les messages de la boîte aux lettres en les regroupant par expéditeur

Représentation interne de la boîte aux lettres

La boîte au lettres peut-être vu comme une file. A chaque fois que l'on lit un message, on l'insère à la fin de notre mémoire sous la forme :

Format de la boîte aux lettres

```
message_received(Message, Sender)
...
message_received(Message, Sender)
```

3.5.6.5 Description des messages

Messages reçus sur les streams de communication

Liste des messages

message sans acquittement : aucun traitement particulier n'est effectué

message avec acquittement : entraîne l'envoi d'un accusé de réception sur la *stream* de service de l'expéditeur

ping(Id) : message de ping externe qui entraîne l'envoi automatique d'un message **pong(Id)** à l'expéditeur

Messages reçus sur les streams de service

Les messages de service sont des messages internes à ALBA qui circulent par les *streams* de service des agents. Cette section a pour objet de dresser la liste des messages de service traités par la

1 Par exemple en cas d'erreur dans le code de l'agent

bibliothèque.

Liste des messages de service

```
alba_stop_system : message de fin du SMA (Cf. Fin des agents)

alba_search : message de recherche d'un agent (Cf. Recherche d'agents)

alba_ping : ping interne pour les messages de service

message_acknowledgement(Id): accusé de réception d'un message (Cf. Envoi de messages)

service_acknowledgement(Id) : accusé de réception d'un message de service
```

Il faut rajouter à ces messages *end_of_file* qui est reçu sur chacune des *streams* lorsque le contact est rompu.

3.5.6.6 Encapsulation des messages

Les messages qui circulent sont automatiquement encapsulés par ALBA pour les raisons suivantes :

- un identifiant unique étant attribué à chaque message envoyé, il est nécessaire de le fournir dans l'encapsulation du message,
- l'encapsulation permet d'éviter que l'utilisateur envoie un message pouvant être interprété comme interne à ALBA,
- bien que nous ayons à disposition le nom de l'expéditeur d'un message puisque nous savons sur quel *stream* il a été lu, nous verrons que, notamment lors de la migration, il se peut qu'un message soit relayé par un agent intermédiaire, dans ce cas nous avons un expéditeur effectif et un expéditeur originel et il est nécessaire de bien les distinguer,
- la distinction entre message sans et avec accusé de réception nécessite d'associer un type (*normal* ou *acknowledgement*) au message.

Voici comment cela se présente en pratique :

Encapsulation des messages

```
alba_message(Message, Sender, Id, Type)

alba_service_message(Message, Sender, Id)
```

3.5.7 Recherche d'agents

Intérêt

Les agents n'étant identifiés que par leur nom, il est nécessaire de mettre en place un algorithme permettant de se connecter à un agent à partir de la seule connaissance de son nom. Cet algorithme confère plus de souplesse et de robustesse au système que la version 1.0 qui identifiait les agents à partir de leur nom, de leur adresse et de leur port. Il paraît, en effet, cohérent de se reposer sur la seule information stable relative à un agent pour l'identifier dans le système à savoir son nom, alors que son adresse et son port peuvent être amenés à évoluer au cours de sa vie (migration...). Ce principe permettrait notamment de tuer un agent et de le relancer sous le même nom sans que cette modification n'ait d'impact sur les anciennes accointances de cet agent.

Implémentation

L'idée de base est triviale, il s'agit pour un agent A cherchant à communiquer avec un agent B de demander à toutes ses accointances si elles ont connaissance de B, si ce n'est pas le cas elles répandent elles même le message de recherche à leurs voisins. Le concept est très proche d'un algorithme de

recherche dans un graphe dont les sommets seraient les agents et dont les arcs matérialiseraient les connexion entre agents. Le problème est toutefois plus complexe car dans notre cas de figure d'une part nous ne sommes pas certains que les arcs soient valides et d'autre part le graphe peut évoluer dynamiquement durant la recherche. L'algorithme paraît bien adapté aux SMA puisqu'il en exploite le caractère distribué. Bien que d'apparence très simple, il pose en pratique de nombreux problèmes, la difficulté étant de limiter au maximum le nombre de messages envoyés, de s'assurer que la vague de messages s'arrête et qu'un agent recherchant un autre agent présent dans sa composante connexe puisse le trouver à coup sûr. Nous l'avons baptisé du nom très imagé d'algorithme de recherche par vague.

Algorithme naïf

Il est évidemment nécessaire de marquer les agents qui ont déjà reçu le message de recherche sous peine de générer des vagues infinies de messages.

Etudions tout d'abord la structure des messages de recherche qui transitent par les *streams* de service et proposons un premier algorithme naïf.

Structure d'un message de recherche

```
alba_search(SEARCHER_IDCARD, CONTACT_TO_FIND, VISITED_CONTACTS)

SEARCHER_IDCARD : IDCARD du contact ayant déclenché la recherche
CONTACT_TO_FIND : nom du contact cherché
VISITED_CONTACTS : contacts déjà visités
```

Algorithme naïf

```
1. Si je suis l'agent recherché
   Alors je me connecte avec l'agent demandeur
2. Sinon je me rajoute à la liste des agents visités (liste tabou)
         je transmets le message à toutes mes accointances qui n'ont pas
         déjà été visitées.
```

Remarquons que nous avons choisi d'arrêter la vague lorsque le contact recherché reçoit le message de recherche, on aurait pu imaginer arrêter la vague une étape avant, décidant que lorsqu'une accointance du contact recherché reçoit le message de recherche elle transmet directement au chercheur l'IDCARD du recherché. Cette méthode ne fait en pratique rien gagner comme nous le verrons et s'avère moins sûre, il se peut en effet que l'accointance du recherché ait une IDCARD non valide du recherché. Il est donc préférable que ce soit l'agent recherché qui se connecte lui-même auprès de l'agent ayant déclenché la recherche.

Bien évidemment cet algorithme fonctionne très bien et répond au problème posé. Il génère cependant beaucoup de redondance dans les messages envoyés. Le pire cas pour cet algorithme en terme de nombre de messages envoyés est le cas où chaque agent est connecté avec tous les autres. Dans ce cas, on peut montrer que le nombre de messages envoyés est au pire cas de l'ordre de n!, où n est le nombre d'agents du système.

Evaluation du nombre de messages envoyés au pire cas

Soit n le nombre d'agents du système. Chaque agent étant connecté à tous les autres, son nombre d'accointances est de n – 1.

Soit M le nombre total de messages envoyés et M(α) le nombre de messages envoyés à l'étape élémentaire α. On en déduit donc, du fait des spécificités de l'algorithme, que la liste tabou comporte α éléments à cette étape. Il s'en suit que le nombre total d'étapes est de n-1.

On a,

$$M = \sum_{\alpha=1}^{n-1} M(\alpha)$$

Posant $M(0)=1$, on a alors,

$$M(\alpha)=\sum_{i=1}^{M(\alpha-1)} n-\alpha$$

On remarque que la somme ne dépend pas de son indice, on obtient donc,

$$M(\alpha)=(n-\alpha)\sum_{i=1}^{M(\alpha-1)} 1 =(n-\alpha)\cdot M(\alpha-1)$$

D'où, par récurrence immédiate,

$$M(\alpha)=(n-1)\cdot(n-2)\cdots(n-\alpha)=\frac{(n-1)!}{(n-\alpha-1)!}$$

Ainsi,

$$M=\sum_{\alpha=1}^{n-1} M(\alpha)=(n-1)!\sum_{\alpha=1}^{n-1}\frac{1}{(n-\alpha-1)!}$$

Soit, par changement de variable,

$$M=(n-1)!\sum_{k=0}^{n-2}\frac{1}{k!}=(n-1)!\left(1+\sum_{k=1}^{n-2}\frac{1}{k!}\right)$$

Or, $e=\lim_{n\to+\infty}\sum_{k=1}^{n}\frac{1}{k!}$ (d'après Taylor), ainsi,

$$(n-1)!\leqslant M\leqslant(1+e)\cdot(n-1)!$$

Et donc,

$$M\in\Theta((n-1)!)=\Theta(n!)$$

Il faut donc absolument améliorer notre méthode.

Anticipation

Nous avons vu qu'il est absolument nécessaire de prendre des dispositions pour limiter le nombre des messages de recherche envoyés. Pour ce faire, il est possible d'exploiter la seule information supplémentaire qui soit disponible au niveau d'un agent à savoir la connaissance de ses accointances afin d'anticiper d'une étape. Ainsi, lorsque A reçoit le message de recherche, il se rajoute à la liste des agents visités, et il rajoute également à cette liste l'ensemble de ces accointances à qui il envoie le message de recherche. Ce regard en avant d'une étape engendre des difficultés comme nous allons le voir.

De la nécessité d'un traitement d'erreur

Illustrons sur un schéma les difficultés qui peuvent apparaître du fait de cette anticipation.

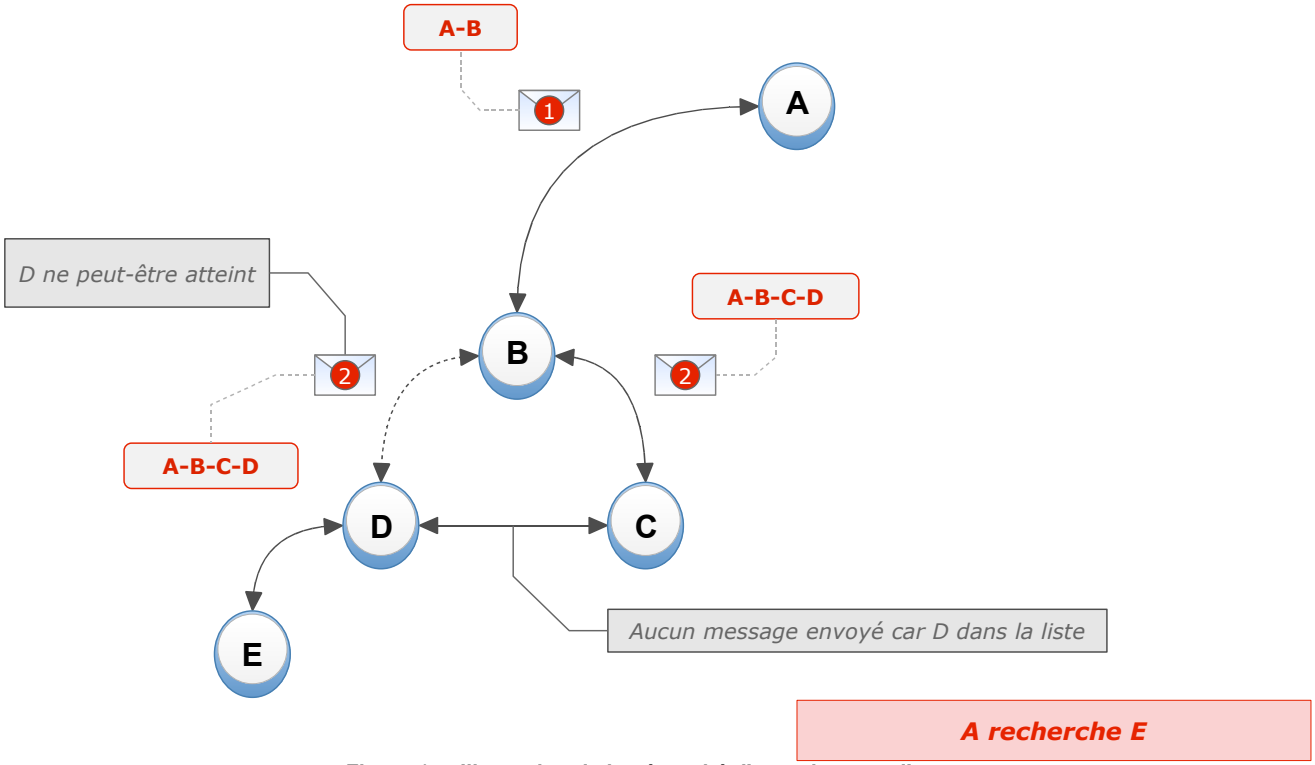


Figure 1.4: Illustration de la nécessité d'un traitement d'erreur

On voit sur cet exemple que A recherche E mais ne peut le trouver, bien qu'ils soient dans la même composante connexe, suite à une erreur dans l'envoi de message de B vers D.

Pour pallier ce problème nous avons pensé mettre en place un système de ping visant à valider les connexions comme illustré sur ce schéma qui reprend l'exemple précédent mais dans le cas où un ping est systématiquement envoyé :

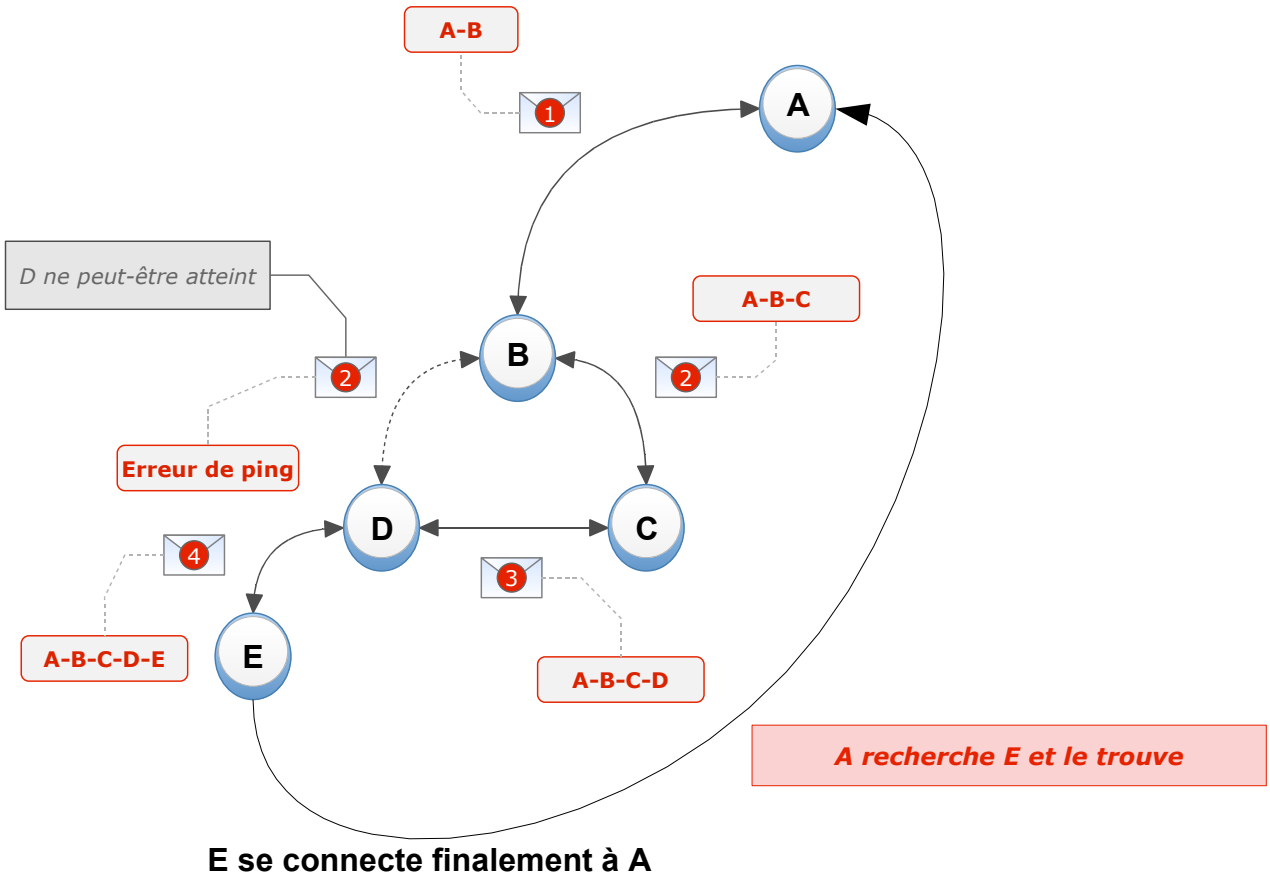


Figure 1.5: Illustration du système de ping

Nous avons estimé finalement que cette solution générait trop de messages et avons donc opté pour l'alternative suivante. Il est en effet coûteux d'envoyer un message de ping à chaque étape et à toutes les accointances pour vérifier que les connexions sont valides alors qu'en pratique elles le sont dans la grande majorité des cas.

En pratique il est possible de tester la validité d'une connexion en envoyant un message. Si le message part correctement sans soulever d'exception, l'arc est valide. Il n'est donc pas nécessaire d'attendre une réponse de l'agent dont on souhaite vérifier le contact. Les cas de rupture de contact sont, en pratique, extrêmement rares mais il faut être capable de supporter une éventuelle erreur de l'envoi de message. Une solution hybride a donc été mise en place. Elle est aussi efficace que la solution sans vérification d'erreurs dans le cas normal et permet de gérer d'éventuelles difficultés lorsqu'elles se présentent.

Le traitement local des erreurs se fait ainsi : lorsque l'agent reçoit un message de recherche il répand la recherche à l'ensemble C de ses contacts (non déjà visités) en les ajoutant aux contacts visités. La routine d'envoi de message nous donne immédiatement la liste des contacts E pour lesquels le message de recherche n'a pu être acheminé avec succès. On renvoie alors un message de recherche à C – E en enlevant E des contacts visités. En pratique la liste est vide est aucun traitement supplémentaire n'est à faire. Nous voyons donc comment cette solution intermédiaire nous permet de résoudre efficacement notre problème.

De l'inutilité d'un appel récursif

Il a été envisagé pendant un moment de faire des appels récursifs à l'algorithme de recherche par vague. Ainsi si un agent A ayant deux accointances B et C cherche à joindre D, sachant que le lien entre A et B est rompu. Des appels à des vagues récursives entraîneraient une recherche par vague de l'agent B dans le but sous-jacent de trouver D. On se rend bien compte qu'il est complètement inutile que A demande à C si il connaît B dans le but de trouver D, car en répandant la recherche par vague visant à trouver D, l'agent C répond déjà implicitement à cette question.

Une autre alternative

Nous avons pensé un temps qu'il serait envisageable de stocker localement les noms des personnes recherchés au sein de chaque agent. A chaque initialisation de connexion entre deux agents ceux-ci pourraient alors s'échanger leurs avis de recherche respectifs et éventuellement en supprimer certains. Dans cette méthode lorsqu'un agent est retrouvé il répand l'information et les autres agents suppriment l'avis de recherche correspondant. On peut voir ce système comme une métaphore du fonctionnement des commissariats. En pratique cet algorithme nous a semblé peu convaincant. Il n'est tout d'abord jamais très sain de sauvegarder des informations susceptibles de rester indéfiniment en mémoire de l'agent et d'occuper de plus en plus d'espace.

Le schéma ci-dessous illustre un autre problème critique de la méthode :

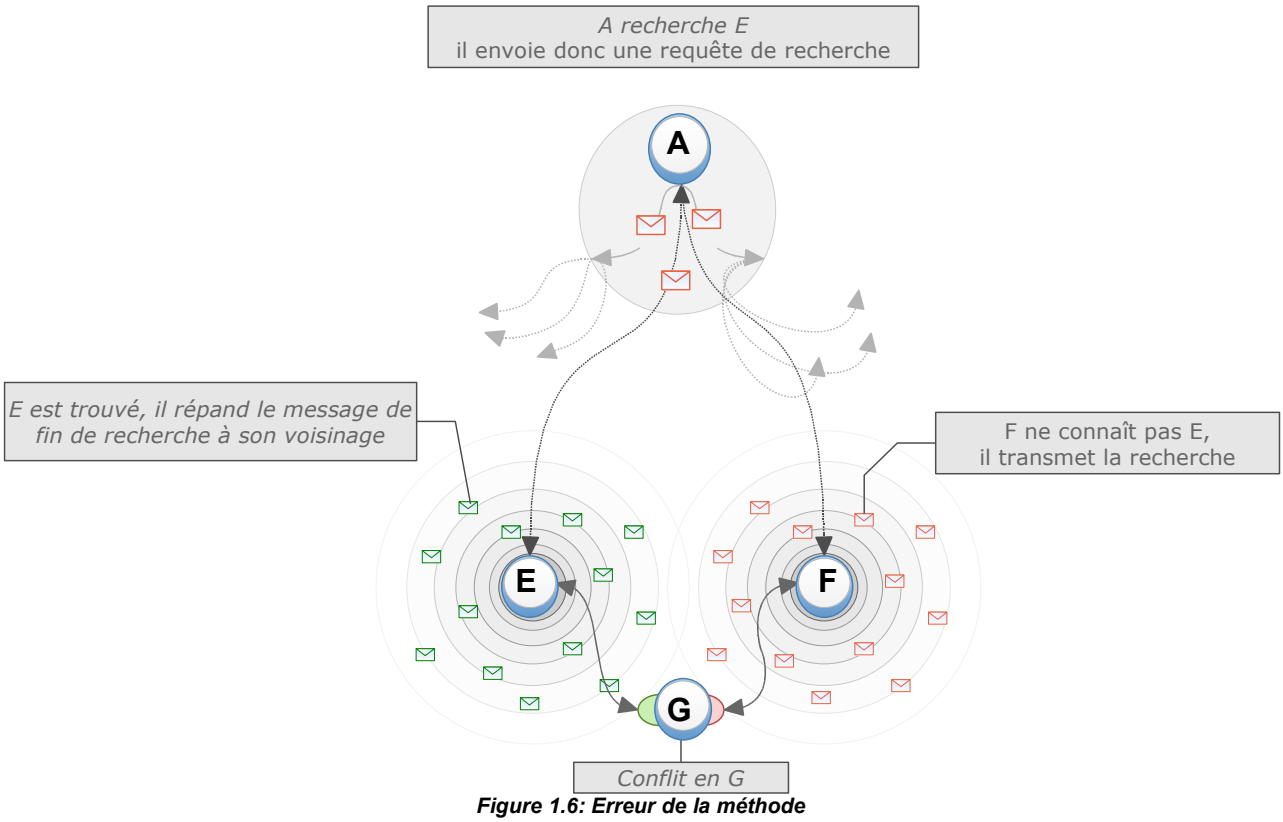


Figure 1.6: Erreur de la méthode

Comme on le voit des conflits peuvent naître lorsqu'un agent reçoit pratiquement en même temps un message de fin de recherche et un message de recherche.

La méthode a donc été abandonnée. Voyons donc l'algorithme qui a été adopté au final.

Algorithme actuel

Voici l'algorithme tel qu'il se présente aujourd'hui :

Algorithme de recherche

```
1. Si je suis l'agent recherché
   Alors je me connecte à l'agent demandeur
2. Si j'ai l'agent recherché dans mes accointances
   Alors je tente de lui transmettre le message de recherche.
     Si échec dans l'envoi je poursuis en 3.
3. Mise à jour du message de recherche initial M en ajoutant l'ensemble des
contacts non déjà visités C à la liste des visités.
4. Envoi du message de recherche mis à jour à C, E nous donne la liste des
contacts n'ayant pu recevoir le message de recherche
5. Si E est non vide
```

```
   Alors modification du message M en ajoutant C - E à la liste des visités.
Envoi du nouveau message à C - E
```

Amélioration envisageable

Bien que la méthode actuelle fonctionne de façon satisfaisante, il serait envisageable d'orienter la recherche en fonction du nom de l'agent recherché. Ainsi si l'on cherche *root/toto/tutu* il est intéressant de se diriger prioritairement vers les ancêtres du *root/toto/tutu* à savoir *root/toto* et *root*. Toutefois, au fur et à mesure de l'évolution du SMA si des agents se terminent ces liens sont susceptibles d'être rompus.

3.5.8 Hétéro-agents

Nous ne reviendrons pas ici sur l'intérêt de pouvoir inclure dans le système des agents hétérogènes et de pouvoir les faire intéragir avec des agents ALBA comme des agents presque « *normaux* ». Voyons comment cela se déroule :

```
merge_external_entity(+Name, +Host, +Port, +Stream)

Name : nom de l'agent hétérogène
Host : adresse de l'agent
Port : port de l'agent
Stream : stream de message associé à l'agent
```

Un appel au prédicat *merge_external_entity* entraîne donc la création d'un contact. Dans ce cas, au lieu d'établir une connexion comme habituellement on se contente d'utiliser *Stream* comme canal de communication. Il est donc possible de communiquer dans les deux sens par cette *stream* avec l'hétéro-agent. Il faut toutefois noter que ces communications se font sans aucune encapsulation des messages dans un sens comme dans l'autre, l'hétéro-agent étant, par nature, complètement indépendant des contingences d'ALBA. Contrairement aux autres agents, un hétéro-agent n'a donc pas de *stream* de service, celle-ci est donc renseignée à *null* dans les informations du contact.

3.5.9 Fin des agents

Afin de fermer proprement un agent, il est souhaitable d'utiliser le prédicat suivant :

```
stop_agent
```

Un appel à *stop_agent* entraîne la fermeture propre de toutes les *streams* ouvertes par l'agent. Cette tâche effectuée, l'agent se ferme par un appel à *halt* qui tue le processus. Il n'y a pas d'envoi explicite de message d'arrêt aux accointances car lorsqu'un agent fera appel à *core_treatments* il recevra *end_of_file* sur la *stream* associé à l'agent mort ce qui lui permettra de nettoyer son contact. Des procédures ont d'autre part été mises en place pour s'assurer de la validité des *stream* utilisées.

Si la fermeture d'un agent ne pose pas de problème particulier, nous nous sommes penchés sur la question de la fermeture globale du système. Il n'est pas question de contraindre l'utilisateur à fermer manuellement chacune des fenêtres correspondant aux agents lancés. En plus d'être fastidieuse, surtout lorsque l'on a des dizaines d'agents lancés, cette méthode ne permet pas de supprimer les agents ouverts sans fenêtre. Pire elle ne traite que les agents du système résidant sur la machine locale. C'est dans cette optique que la routine suivante intervient :

```
stop_all_agents
```

Le fonctionnement de cette routine est très simple. L'agent qui déclenche un *stop_all_agents* envoie à tous ses contacts un message *alba_stop_system* sur leur canal de service, il ferme ensuite tous ses *streams* et se tue. Chaque agent recevant le message *alba_stop_system* le fait suivre à tous ses contacts, excepté bien sûr à l'expéditeur, et procède de même. Le système s'éteint ainsi immédiatement par une

vague de messages de clôture qui est répercutée de proche en proche. A supposer que le système constitue une seule et même composante connexe ce qui est généralement le cas dans les SMA que l'on utilise puisque les créations d'agents sont hiérarchiques et que la densité des connexions permet de compenser une éventuelle rupture de lien, l'extinction du système est complète. Il peut toutefois arriver qu'un agent soit pris dans un long calcul auquel cas le processus se terminera une fois cette tâche effectuée.

Le problème de la mort des agents n'a toutefois pas été totalement résolu et des investigations plus poussées mériteraient d'être menées à ce sujet. A l'heure actuelle, nous ne stockons pas d'informations sur la mort des agents. Ceci se justifie par plusieurs arguments :

- nous voulons un maximum éviter de stocker des informations qui peuvent potentiellement s'amonceler au fur et à mesure de l'évolution du SMA. Un SMA doit en effet pouvoir tourner sur de longues périodes sans planter
- nous ne voulons pas centraliser les informations or il semble que la liste des agents morts nécessite une forme au moins partielle de centralisation

Cette façon de procéder à une influence directe sur la recherche par vagues des agents, à chaque fois que l'on recherche un agent absent du système, une nouvelle vague de messages est en effet déclenchée. Cette recherche inutile pourrait être évitée pour des agents dont l'on sait qu'ils sont morts et qu'il serait donc définitivement vain de les rechercher encore et encore.

3.5.10 Migration

```
migrate(+Host, +Window)

Host : hôte vers lequel l'agent doit migrer
Window : avec ou sans fenêtre
```

3.5.10.1 Schéma de la mémoire

Avant de développer l'algorithme de migration, il est nécessaire d'avoir une bonne vue d'ensemble de l'organisation mémoire des données internes d'ALBA. En voici un aperçu pratiquement exhaustif, nous avons marqué d'un * les informations devant impérativement être transmises lors de la migration de l'agent pour assurer une bonne continuité d'exécution suite à la migration.

Vue de l'organisation mémoire des données internes

```
% Contacts
contact(Name, idcard(Name, Host, Port), MsgStream, ServiceStream, Status)*
...


% Messages
message_received(Message, Sender)*
...

% Pending Messages
message_pending(Recipient, MsgPending, NbMsg, ServPending, NbServ)*
...

% Name used
name_used(Name, Id)*
...

message_id(Id)*

service_message_id(Id)*

personal_infos(IdCard, Behavior, Stream)

migration_luggage(L)*
```

3.5.10.2 Algorithme de migration

Principes

Une migration peut-être considérée comme réussie si elle est transparente et imperceptible, en terme de comportement des agents, pour un observateur extérieur.

Le principe général de l'algorithme peut-être décrit comme suit :

- le migrant A lit tous ses messages et crée un clône de lui-même sur la machine cible. Pour ce faire, il lui transfert son code source, ses fichiers de travail, ainsi que toutes les informations que nous avons marqué d'un * dans le section précédente
- le clone s'identifie auprès de toutes les accointances du migrant comme le nouvel agent A
- le migrant de son côté se contente de renvoyer les messages qu'il reçoit à son clone. Il reste actif jusqu'à ce qu'il n'ait plus de messages ni d'agent fils en phase de création.
- Le migrant informe alors son clone qu'il va se détruire. Le clone prend alors le nom de A et commence à s'exécuter

Voici un schéma exprimant graphiquement ces étapes :

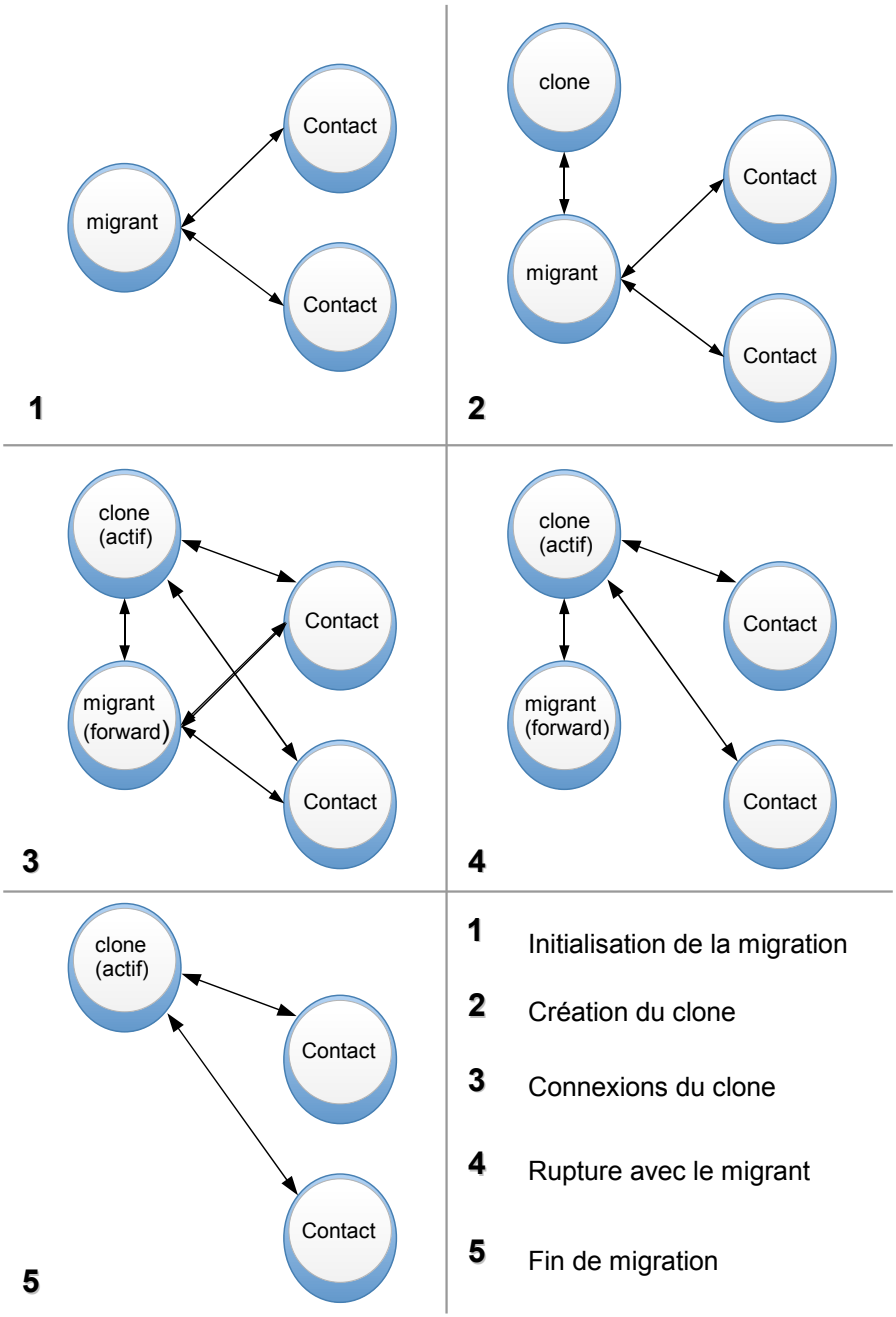


Figure 1.7: Protocole de migration

Etude plus détaillée

Disposant d'une vue très générale du fonctionnement de l'algorithme, il nous faut maintenant décrire plus précisément comment ces étapes se déroulent pour chacun des acteurs intervenant plus ou moins directement dans la migration. Ce sera l'occasion de comprendre qu'il est complexe de tester précisément tous les cas de figure susceptibles de se produire durant une migration d'agent.

Du côté du migrant

Le processus se déroule en deux étapes.

Création du clone

```
1. traitement des messages de service et lecture de tous les messages
2. création du clone qui entraîne le passage des accointances et le
transfert des fichiers de comportement
3. envoi des données utiles marquées d'un * par un message avec accusé de
réception
4. passage en phase transitoire
```

Phase transitoire

```
1. Faire
     - gérer les nouvelles connexions, à chaque nouvelle connexion un
message est envoyé au clone afin que ce dernier se connecte avec l'agent
voulant rentrer en contact
     - retransmettre les messages reçu au clone
     - retransmettre les messages de service au clone

   Jusqu'à ce qu'il n'y ait plus de nouveaux messages à traiter et plus
d'agent en cours de création
2. Envoi du message de fin de migration au clone
3. Appel à stop_agent
```

Les messages retransmis au clone sont encapsulés proprement afin de faire apparaître l'expéditeur réel du message. Dans le cas contraire, le clone interpréterait tous les messages comme émanant directement de l'agent migrant.

En ce qui concerne les messages de service ils peuvent tous être retransmis au clone sauf bien sûr les éventuels messages d'acquittement qui n'ont pas besoin d'être envoyés au clone. Un message *stop_system* sera en outre traité de manière classique et entrainera l'arrêt de tous les agents migrants, clone inclus.

Du côté du clone

De même, pour le clone le processus de migration se déroule en deux phases.

Initialisation du clone

```
1. initialisations
2. connexion avec le migrant, si échec la procédure est annulé et le clone
détruit
3. connexion aux accointances du migrant par un message spécial leur
spécifiant qu'il émane d'un agent en migration
4. rapatriement des données du migrant
5. traitement des messages provenant du migrant
6. fin de la phase transitoire à la mort du migrant et passage à la reprise
de l'exécution
```

Reprise de l'exécution

```
1. changement de nom, reprise du nom du migrant
2. appel à la routine restart
```

```
restart

prédicat appelé lorsque la migration est achevée pour reprendre
l'exécution du comportement de l'agent. Ce prédicat est défini par le
programmeur d'agent.
```

Le clone ayant été créé par le migrant, il porte au début un nom de la forme *migrant/clone(X)*. Il se fait bien sûr passer pour le *migrant* auprès de ses accoitances et seul le migrant le connait sous ce nom temporaire. Dès que la migration s'achève, le clone met à jour ses informations personnelles et s'approprie l'identité de son père.

Du côté des accointances du migrant

Pour les autres agents du système, la migration est pratiquement transparente. Il leur faut simplement pouvoir être capable de gérer le message de connexion du clone leur spécifiant que c'est dorénavant avec lui qu'ils devront traiter.

Reprise de l'exécution

L'une des difficultés majeurs de la migration réside dans la reprise du comportement de l'agent suite à une migration. Dans l'idéal, l'agent doit reprendre précisément où il en était avant la migration. Dans cette perspective, il est nécessaire d'adapter les outils de description du comportement des agents afin qu'ils soient pleinement compatibles avec la migration. Dans le cadre des fils de raisonnement, il est par exemple indispensable de restaurer la mémoire et l'état courant. La bibliothèque met donc à la disposition du programmeur les deux prédicats suivants qui peuvent être appelés à tout moment de l'exécution d'un agent :

```
put_into_luggage(+Variable, +Value)

Variable : nom de la variable
Value : valeur associée

Ce prédicat permet de mémoriser un nom de variable et de l'associer à une
valeur. Il permet donc de sauvegarder toutes les données utiles en
prévision d'une future migration. La valeur des variables peut-être
écrasée à loisir.
```

```
get_from_luggage(+Variable, ?Value)

Variable : nom de la variable
Value : valeur associée

Ce prédicat permet d'accéder aux valeurs des variables ayant été
sauvegardées. Il permet de restaurer les données d'intérêt après une
migration
```

Améliorations

En l'état la migration fonctionne de manière satisfaisante. Des améliorations restent toutefois à mettre en place. Pour l'instant, le clone est créé à partir du code original de l'agent. L'une des caractéristiques du langage Prolog est qu'il est extrêmement facile pour un processus de modifier dynamiquement son propre code, pour en tenir compte il faudrait donc créer le clone à partir d'une copie du code du migrant dans l'état dans lequel il se trouve en mémoire lors de la migration.

Il est nécessaire d'assurer la robustesse de la migration. Un effort a donc été mené en ce sens, ainsi si la migration échoue en plein processus, la migration est abandonnée et le migrant poursuit sa tâche. Les traitements actuels restent cependant perfectibles et il sera donc souhaitable de poursuivre en ce sens et de faire des tests plus approfondis afin de mieux éprouver le système.

3.5.11 Proto-agents

Actuellement, le concept de proto-agent n'a pas été complètement implémenté. Une réflexion de fond a toutefois été faite sur sa mise en place imminente dans le coeur d'ALBA. Nous allons en présenter ici les grandes lignes.

A l'heure où j'écris ces lignes, les a-agents sont créés à partir de fichiers de comportement (code source, fichiers utiles...) qui sont considérés comme des proto-agents. Plusieurs a-agents peuvent être générés à partir d'un même proto-agent qui constitue donc un moule servant de base aux a-agents. L'utilisation courante est limitée dans le sens où au démarrage chaque a-agent se voit attribué comme espace de travail le répertoire contenant le comportement. Ils n'ont donc pas d'espace propre de travail et la

modification de leurs fichiers de ressource se répercute sur tous les a-agents en cours d'exécution.

Nous souhaitons mettre en place le fonctionnement suivant. A chaque création d'un a-agent à partir d'un proto-agent, le nouvel a-agent créé se voit attribué un espace personnel de travail dans lequel sont copiées toutes les données constituant le corps du proto-agent. Toutes les modifications faites par l'a-agent dans son espace de travail lui sont donc exclusivement personnelles et n'influent pas sur le proto-agent.

L'a-agent se voit ainsi attribué des données rémanentes qu'il peut modifier à loisir. Nous avons prévu d'offrir au programmeur la possibilité d'empaqueter à tout moment dans une archive le répertoire de travail de l'a-agent. Ce dernier pourra ainsi reprendre ses tâches et travailler sur ses données dans l'état dans lequel il les avait laissé avant sa désactivation ou avant une fermeture voulue ou indésirable du SMA.

Lors d'une migration, c'est le répertoire de travail de l'a-agent qui sera archivé et transmis sur la machine distante. La migration pose d'autres problèmes en ce qui concerne la localisation des proto-agents. Ainsi, si l'on se place dans un a-agent A créant dans son code un autre a-agent B à partir du proto-agent b dont le chemin est indiqué sur la machine locale. Si A migre avant la création de B il est bien évident qu'en l'état le proto-agent ne pourra pas être localisé[1]. Les proto-agents seront donc identifiés en interne par l'adresse de la machine où il réside ainsi que par leur chemin complet sur cette machine. Le programmeur désirant créer un a-agent utilisera lui un simple nom, une correspondance entre ce nom et l'identification interne du proto-agent sera ensuite automatiquemet effectuée par le système. On peut rapprocher ce procédé du comportement adopté par java pour rechercher des *classes* dans des chemins définis.

Des réflexions sont également en cours, sur la manière de limiter un maximum les transferts de proto-agents d'une machine à l'autre.

3.5.12 Gestion des erreurs

La gestion des erreurs a retenu toute notre attention. Cela s'est traduit par différentes dispositions :

- afin d'assurer la meilleure robustesse possible à notre bibliothèque de nombreux tests de cohérence des entrées fournies sont effectués dans toutes les routines,
- lorsqu'une erreur est identifiée, nous affichons un message destiné à l'utilisateur du SMA. Un travail a été fourni pour que ces messages soient le plus explicites possible. Dans tous les cas, la bibliothèque tente de poursuivre l'exécution autant que faire se puisse,
- un système d'exception a été mis en place en interne, les exceptions systèmes ont été encapsulées et de nouvelles exceptions propres à ALBA ont été défini,
- des coupures ont été placées à la fin des routines offertes au programmeur d'agent afin d'éviter des retours en arrière non désirés

Lorsque beaucoup d'agents sont lancés en même temps, potentiellement sur plusieurs machines, il est très mal aisé pour un observateur de voir les messages d'erreurs susceptibles d'apparaître. Il est possible de pallier ce problème grâce à la bibliothèque que nous allons décrire ci-après.

3.5.13 Bibliothèque d'évènements

Description de la problématique

Comme nous l'avons mentionné précédemment (Cf. SpySMA, Phenix pour plus de détails), il était nécessaire d'assurer la compatibilité d'ALBA 2.0 avec SpySMA. Nous avions également vu que l'intégration de SpySMA dans le code d'ALBA n'était pas satisfaisante.

La première solution envisagée fut de définir un certain nombre d'évènements dans ALBA et de permettre au programmeur d'appeler des prédicats lorsque ces évènements étaient levés. Cette solution était fonctionnelle mais peu puissante, elle impliquait d'écrire à la main des appels de prédicats dans de nombreuses routines d'ALBA. Nous avons donc tenté de généraliser cette idée afin d'offrir une bibliothèque générique d'évènements pouvant être utilisée dans tous les développements Prolog du service et ne nécessitant pas d'écrire de code supplémentaire dans toutes les routines utilisant des évènements.

1 À moins bien sûr qu'il ne soit également présent au même endroit sur la nouvelle machine de A, ce qui n'est généralement pas le cas, le programmeur ne peut et ne doit évidemment pas se reposer sur de telles hypothèses

Implémentation

Nous présentons ici la bibliothèque telle qu'elle a été implémentée pour l'instant, nous verrons que les perspectives d'évolutions sont nombreuses et très intéressantes.

La bibliothèque d'évènements offre trois prédicats principaux dont la description va nous éclairer sur son mode de fonctionnement. Nous l'appréhenderons encore mieux en la mettant en pratique sur l'exemple d'ALBA.

Le premier prédicat permet de définir un évènement. Les définitions d'évènements se placent en début de fichier source désirant utiliser la bibliothèque.

```
define_event(+Event, +Predicates, +Args)

Event : nom de l'évènement à définir
Predicates : liste de prédicats déclenchant l'évènement.
Args : liste de liste d'arguments. Chaque prédicat est associé à une liste
d'entier correspondant aux indices des arguments qui présentent un intérêt
pour l'évènement
```

Il est ensuite possible de définir des *handlers* depuis des fichiers extérieurs qui seront appelés lorsque les évènements seront déclenchés. Les *handlers* sont automatiquement déclenchés avec comme premier paramètre la date de déclenchement, et comme autres paramètres ceux décrit par l'argument Args de *define_event*. Un même évènement est donc associé à une arité fixe qui sera la même pour tous les *handlers* associés. Des tests de cohérence sont évidemment effectués. Il est bien sûr possible de définir plusieurs évènements utilisant le même prédicat, et d'associer plusieurs prédicats à un même évènement.

```
add_handler(?Event, +Predicate)

Event : nom de l'évènement à associer au handler. Si Event est libre le
handler sera associé à tous les évènements du système
Predicate : handler à associer avec Event
```

```
remove_handler(?Event, ?Predicate)

Event : nom de l'évènement, si Event est une variable, le handler
Predicate est supprimé pour tous les évènements
Predicate : nom du handler à désactiver. Si Predicate est une variable,
tous les handlers associés à l'évènement sont supprimés.
```

Voyons comment cela se déroule en pratique. La définition d'évènements en tête de fichier provoque l'introduction en mémoire de règles qui se chargeront d'instrumenter automatiquement le code. Le code des prédicats déclencheurs d'évènements sera ainsi automatiquement modifié pour appeler tous les *handlers* associés, avec les arguments voulus, lorsqu'une clause du prédicat est achevée avec succès. Cette modification dynamique se fait grâce au prédicat Prolog *term_expansion* qui permet précisément de modifier des clauses s'unifiant avec un motif donné.

Les appels à *add_handler* et *remove_handler* se contentent d'ajouter et de retirer en mémoire les *handlers* associés aux évènements.

On voit donc comment, par ce procédé, il devient possible de laisser le programmeur injecter n'importe quel code à des endroits précis et soigneusement choisis du code source d'origine. Nous verrons ci-après les perspectives que cela peut offrir.

Application à ALBA

Voyons maintenant comme nous pouvons utiliser cette bibliothèque au sein d'ALBA. Il faut d'abord définir un certain nombre d'évènements que l'on souhaite utiliser pour ALBA, en voici une liste indicative :

Evènements ALBA envisagés

```
onInitialization
onSystemStop

onAlbaException
onAlbaMemoryRead
onAlbaMemoryWrite

onAgentCreation
onAgentDestruction
onAgentWaiting
onAgentSearch
onAgentMigration


onMessageRead
onMessageSent
...
```

Le code d'ALBA étant modifié comme suit :

Code d'ALBA modifié

```
Définition des évènements
ex : define_event(onMessageRead, [read_message/3], [1,2])

Code d'ALBA
```

N'importe quel fichier externe peut ensuite associer des *handlers* personalisés aux évènements ALBA. Ainsi le code de SpySMA serait très simplifié et ressemblerait à cela :

Code de SpySMA

```
add_handler(onMessageRead, TraitementReadMessage)
....

TraitementReadMessage(Time, Message, Sender) :- ...
```

La désactivation de SpySMA se fait par un simple *remove_handler(_, _)*.

On voit comment cette bibliothèque générique d'évènement nous permet de répondre aux problèmes qui s'étaient posés pour SpySMA :

- le code de SpySMA est simplifié et aucun appel explicite à SpySMA n'est fait au sein d'ALBA
- nous n'avons rien eu à modifier dans le code d'ALBA

Evolutions envisagées

Si l'exemple de SpySMA est révélateur, il est possible d'envisager bien d'autres applications pour cette bibliothèque au sein d'ALBA. Elle permet aux programmeurs tiers de développer leurs propres extensions à ALBA sans toucher à son coeur. Une extension envisagée pourrait par exemple consister en la mise en place d'un système de cryptographie asymétrique qui permettrait d'assurer la confidentialité des messages échangés en milieu non contrôlé ainsi que l'authenticité des agents avec lesquels on communique. Pour répondre au problème posé dans la section précédente (Cf. Gestion des erreurs), il serait possible d'associer un agent d'interface chargé d'afficher des fenêtres d'erreur associées à l'évènement onAlbaException. Nous verrons que d'autres applications sont également possibles dans le cadre de la thèse de Mlle Caroline Chopinaud nous en reparlerons.

Il faut avouer que notre bibliothèque, qui a pour l'instant plutôt servi de test de faisabilité grandeur nature, est très perfectible et peut être encore généralisé. Il faut notamment en améliorer grandement la syntaxe qui n'est pour l'instant pas très esthétique et peu conforme aux habitudes de Prolog.

Il serait également très intéressant de pouvoir associer des *handlers* sur toutes les portes de la boîte d'exécution de Prolog dont le schéma suivant rappelle la structure :

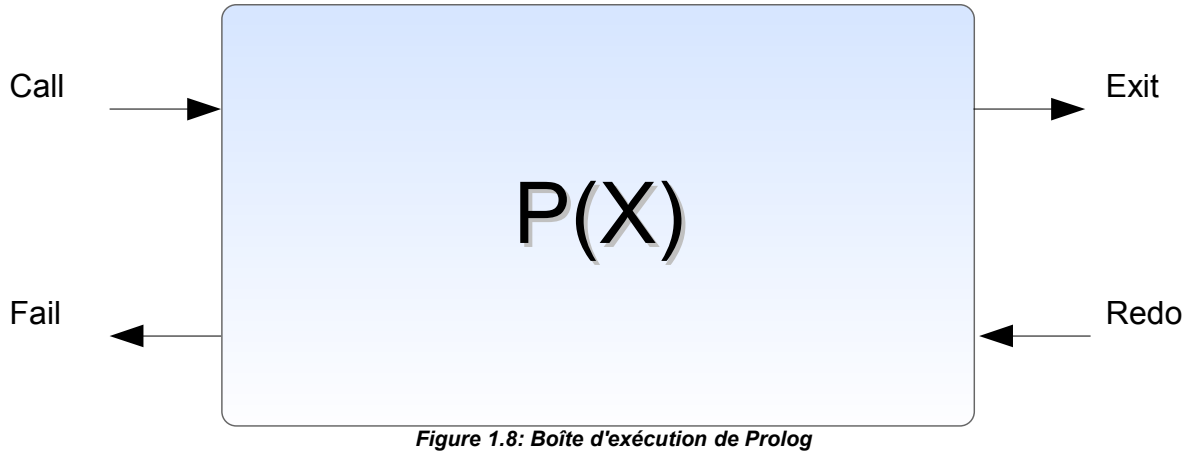


Figure 1.8: Boîte d'exécution de Prolog

Une étude doit être menée à ce sujet pour définir s'il est envisageable, dans le contexte des applications de l'équipe, d'exploiter les outils de « *debugging »* avancés offert par SICStus. Nous ne voulons pas être contraints de subir les ralentissements d'exécution liés au passage en mode « *debug* », nous voulons d'autre part toujours pouvoir compiler des « *runtimes* » fonctionnant sans restriction avec cette bibliothèque. Le manuel de Prolog étant relativement allusif sur ces questions ces investigations nécessiterons un investissement personnel qui n'a pas encore été mené à terme à l'heure où j'écris ces lignes. La bibliothèque est donc encore en état embryonnaire mais semble pleine de promesses.

3.6 Travaux annexes

En sus des travaux qui ont été menés sur ALBA et que nous venons de décrire, des développements annexes furent également réalisés durant le stage.

3.6.1 PrologDoc

3.6.1.1 Motivations

De nombreux codes sources du service sont peu ou mal documentés. Des générations de stagiaires se sont succédées utilisant des notations disparates. J'ai donc pris l'initiative de développer un outil similaire à javadoc mais pour du code source en Prolog et écrit en Prolog.

Les objectifs principaux étaient les suivants :

- homogénéiser les conventions utilisées pour les commentaires de code source
- offrir aux utilisateurs des bibliothèques une documentation centralisée, claire, agréable à lire et efficace à parcourir
- permettre au programmeur travaillant sur un vaste projet de retrouver très rapidement le code d'un prédicat et le fichier dans lequel il se trouve

3.6.1.2 Histoire

Une rapide recherche de l'existant me fit découvrir un outil similaire du nom de PrologDoc. Je m'aperçus bien vite que cet outil était extrêmement rudimentaire. Il ne permettait que de documenter des fichiers Prolog contenus dans le répertoire de l'exécutable PrologDoc, le code HTML généré n'était pas conforme aux standards, le résultat visuel était peu avenant, il n'était pas exempt d'erreurs, n'était pas résistant aux erreurs de syntaxe et ne présentait pas toutes les fonctionnalités souhaitées. Il a donc été nécessaire de recoder pratiquement l'ensemble de l'outil en ne conservant que le coeur que nous avons dû

malgré tout modifier.

Pour information, PrologDoc fut créé à l'université d'Israël en 1999 par :

- Elisheva Bonchek
- Sara Cohen

3.6.1.3 Description

PrologDoc permet de générer automatiquement une documentation au format XHTML à partir de code source Prolog. PrologDoc pourra générer une documentation complète de tous programmes Prolog, il est également possible de documenter le code de manière plus précise en utilisant une syntaxe propre à PrologDoc afin d'obtenir des résultats plus satisfaisants.

PrologDoc est sous licence GPL

3.6.1.4 Développement

Le développement de PrologDoc représente un investissement conséquent mais qui s'est réparti sur une longue période par intermittence. Il a été réalisé en étroite collaboration avec mes collègues Silvère Lhermite et François Chazal que je remercie vivement. Silvère Lhermite s'est occupé majoritairement du *pretty_printer* et François Chazal de l'interface java (Cf. Annexes pour un aperçu). Ils ont, d'autre part, tous les deux énormément contribués tout au long du développement de l'application par leurs commentaires avisés.

Nous passons ici l'ensemble des détails implémentatoires qu'il a fallu aborder pour atteindre le niveau de qualité que nous souhaitions car il serait beaucoup trop long de les aborder tous et cela ne présenterait sans doute que bien peu d'intérêt pour le lecteur. Disons simplement que ce développement a nécessité une connaissance très approfondi des détails du langage Prolog.

Un exemple de résultat produit par PrologDoc pour la documentation d'ALBA est fourni en Annexes.

3.6.1.5 Utilisation

L'utilisation se fait soit par le biais d'une interface Java dans laquelle on choisit :

- le répertoire à documenter, sachant que le programme descend récursivement dans les sous répertoires et documente tous les fichiers .pl rencontrés,
- le répertoire de destination de la documentation
- un fichier facultatif définissant d'éventuels opérateurs définis par l'utilisateur
- il est possible de choisir si l'on souhaite documenter tous les prédicats (documentation destinée à un développeur du projet documenté) ou bien simplement les prédicats exportés (documentation plutôt destinée à un programmeur simple utilisateur du projet documenté)

Il est également possible de documenter un projet à partir d'un simple appel à un prédicat Prolog pour les réfractaires au Java.

Informations générées automatiquement

PrologDoc génère automatiquement un certain nombre d'informations sur votre code source sans que vous ayez à changer quoi que ce soit dans celui-ci. Notons tout d'abord que PrologDoc génère un fichier HTML par fichier .pl fourni. On trouve dans ces fichiers toutes les informations concernant chaque fichier .pl correspondant. PrologDoc génère également un *index.html* qui centralise la liste des fichiers documentés et des prédicats qu'ils contiennent. Cela permet d'avoir une vue globale du projet documenté et d'accéder rapidement à la documentation d'un fichier ou d'un prédicat. Notons qu'une distinction est faite entre les prédicats du même nom.

Voici la liste des informations générées automatiquement :

- Nom du fichier documenté et éventuellement du module qu'il définit

- une liste des erreurs syntaxiques rencontrées dans le code source
- la liste des bibliothèques SICStus utilisées
- la liste des modules utilisés
- la liste des fichiers consultés
- la liste des effets de bord
- la liste des prédicats définis dans le fichier

Documentation générale d'un fichier

Il est possible d'améliorer la documentation générée en ajoutant n'importe où dans le fichier un commentaire de description générale qui doit avoir la forme suivante :

```
/***
@descr description générale du fichier
@author auteur1
@author auteur2
...
@date date
*/
```

Documentation d'un prédicat

Il est possible de documenter chaque prédicat de la manière suivante :

```
/**

...

*/
```

Les *tags* utilisables sont :

- **@descr** : donner une description générale du prédicat
- **@form** : pour donner la forme générale
- **@constraints** : pour donner des contraintes aux paramètres. Ce *tag* doit être suivi de
  - **@ground** : paramètre instancié
  - **@unground** : paramètre non instancié
  - **@unrestricted** : paramètre instancié ou non (sans contrainte)

Exemple :

```
/**
   @form member(Value, List)
   @constraints
      @ground Value
      @unrestricted List
   @constraints
      @unrestricted Value
      @ground List
   @desc True if Value is a member of List
*/
```

Modification de la documentation générée

La documentation générée étant conforme aux standards *XHTML 1.0* et *CSS* du W3C, il est extrêmement simple de modifier complètement la présentation du document généré et de l'adapter à tous les besoins.

3.6.1.6 Perspectives

Nous sommes particulièrement satisfaits de la qualité et de la maturité à laquelle s'est hissé PrologDoc. L'outil peut s'avérer fort pratique pour retrouver des prédicats parmi une masse importante de fichiers et il est même possible de consulter le code source en ligne dans le document avec une coloration syntaxique et une indentation propre. Nous envisageons d'ailleurs de le fournir à SICStus pour qu'il soit intégré dans la distribution.

Bien sûr l'outil reste perfectible, il serait par exemple intéressant de proposer un export en XML. Il reste quelques « *bugs* » mineurs. Il serait également souhaitable d'améliorer la syntaxe des commentaires.

3.6.2 Organisation des projets

Durant le stage, un serveur CVS a été installé. A notre arrivée, les différents projets étaient éparses et la mise en place des outils, bibliothèques et applications, dont certaines sont dépendantes les unes des autres, nécessitait un certain investissement. Nous avons donc tenté de rationaliser les choses afin de se donner un cadre, adapté à la philosophie de CVS, pour organiser les futurs projets.

Voici les normes qui ont été adoptées :

- à la racine de chaque projet se trouve un fichier présentant une description du projet et ses différents prérequis,
- les répertoires séparent distinctement le code source (avec au niveau du dessous un répertoire par langage utilisé), les ressources, les documents...
- un script central fait appel à des sous-scripts définis par module, il est possible d'écraser les variables du script central au sein de chaque module

Ce système permet de rapatrier les différents modules applicatifs du service et de les utiliser directement. Cela s'avère très appréciable pour les nouveaux stagiaires qui pourront immédiatement utiliser les applications sans aucune difficulté.

Un effort a également été consenti pour permettre le passage des anciens projets vers ce nouveau cadre.

Le système actuel reste toutefois perfectible. Il conviendrait de l'améliorer pour lui conférer plus de souplesse notamment lorsque l'on souhaite utiliser concurrement différentes versions des mêmes applications qui ont chacune des prérequis distincts.

Chapitre 4
Bilan du Stage

Nous espérons avoir convaincu le lecteur de la cohérence des démarches qui ont été menées tout au long de cette expérience. Le temps est venu de dresser le bilan de ces six mois. Nous tenterons, dans un premier temps, d'évaluer l'intérêt objectif qu'a pu présenter ce stage pour l'entreprise. Nous en dégagerons ensuite les enseignements personnels, tant sur un plan humain que technique. Il s'agira également de porter un regard rétrospectif et critique sur le travail fourni. Nous conclurons enfin sur les perspectives offertes par le projet.

1.1 Intérêt du stage pour l'entreprise

Il serait prétentieux et illusoire que de croire que les travaux qui ont fait l'objet de ce stage puissent influencer la vie de l'entreprise. Il nous est cependant permis de penser qu'ils ont contribué et contribueront à l'avancée des recherches et applications du service.

ALBA

A l'heure actuelle ALBA offre un socle générique, efficace, flexible, décentralisé et robuste de déploiement des SMA. La bibliothèque sert ainsi déjà de support à de nombreux projets du service :

- Interloc : logiciel de localisation passive de cibles marines
- Demeter : plate-forme au sens propre du terme qui permet notamment à un opérateur humain d'initialiser un SMA, de lancer des agents à la main... (Cf. Annexes pour un aperçu)
- Fils de raisonnement

Le développement d'ALBA a également permis d'affiner et de réaffirmer les concepts généraux des SMA tel qu'ils sont conçus dans le service. La bibliothèque semble ainsi pouvoir servir de base saine et cohérente à tous les futurs projets du service qui nécessiteront de faire intervenir des SMA. Elle pallie aux défauts de la précédente mouture et répond au cahier des charges et aux attentes initiales de mon maître de stage.

PrologDoc

PrologDoc est également un apport de ce stage qui pourra contribuer à la qualité et à l'efficacité de développement des futurs projets du service. Il pourra également jouer un rôle non négligeable dans le transfert des connaissances entre les générations de stagiaires ainsi que dans une meilleure prise en main des outils Prolog qui ont été développés.

Organisation des projets

Bien que perfectible, l'organisation des projets que nous avons mise en place a déjà permis de rationaliser le mode de fonctionnement de l'équipe, elle a d'autre part été intéressante et utile dans le sens où elle a déclenché des réflexions de fond sur la question. Ces investigations ont d'ailleurs toujours court à l'heure où j'écris ces lignes.

1.2 Intérêt personnel

Plan humain

D'un point de vu humain, cette expérience a été extrêmement enrichissante. Voici les principaux aspects que je souhaite mettre en avant :

- développement de ma capacité à exprimer et à défendre mes idées en les argumentant, cela s'est avéré notamment utile pour défendre la pertinence de certains choix techniques d'ALBA auprès de ses utilisateurs souvent enclin à vouloir ajouter des fonctionnalités dans le coeur de la bibliothèque alors qu'elles n'ont en fait pas leur place au niveau où l'on se place. L'une des difficultés du stage résidait d'ailleurs dans la nécessité de bien distinguer ce qui devait faire l'objet de la bibliothèque et ce qui se situait à un autre niveau de granularité ou d'abstraction
- autonomie
- il a été très enrichissant de pouvoir débattre et échanger sur tous les domaines avec mes collègues. Il est d'ailleurs fort intéressant de pouvoir collaborer avec des étudiants d'autres formations
- je suis également ravi d'avoir pu nouer de nouvelles relations dans mon domaine de compétence et d'avoir appris à mieux connaitre deux camarades de promotion

Compétences techniques

Sur un plan plus technique, le stage a également été riche d'enseignements. J'ai notamment pu apprendre et utiliser de manière relativement poussée la programmation logique et plus spécifiquement le langage Prolog. J'ai pu approfondir mes connaissances sur les SMA et me convaincre de la pertinence de leur utilisation. Il m'a été donné d'acquérir une bonne expérience de la programmation en environnement distribué et des difficultés qu'elle peut poser.

J'ai également pris conscience qu'il est fondamental de définir soigneusement les concepts que l'on manipule et de les généraliser au maximum ce qui assure la bonne cohérence des outils développés.

1.3 Conclusions

1.3.1 Perspectives d'évolution

ALBA

Il est clair qu'une bibliothèque comme ALBA ne peut jamais être vraiment considérée comme achevée, elle restera toujours perfectible et évoluera sans doute à l'avenir avec les besoins du service, voici quelques points qui doivent encore être développés et améliorés[1] :

- Il faut tout d'abord poursuivre les travaux sur la migration des agents. Il s'agit principalement de s'assurer que les agents puissent reprendre leur travail là où il s'était interrompu. Ces investigations joueront sans doute un rôle substantiel dans une perspective de *réparation en ligne des agents*. Le but serait à terme de pouvoir *réparer* un agent défectueux en le remplaçant par un agent viable de manière complètement dynamique et transparente pour son environnement.
- Un effort doit également être investi dans une mise en place plus avancée du concept de proto-agent au sein d'ALBA. Il conviendra d'aborder cet aspect avec soin car il aura une incidence forte sur les développements du service. A cet égard, il est peut-être regrettable, a posteriori, de ne pas avoir suffisamment approfondi dès le début ce concept.

1 Nous espérons pouvoir en traiter le plus possible avant la fin du stage

- L'amélioration de la bibliothèque générique d'évènements semble également être un point essentiel à développer. Comme nous l'avons mentionné, elle permettra d'injecter du code en des points centraux d'ALBA et ainsi ajouter de nouvelles fonctionnalités sans retoucher au coeur de la bibliothèque. Cela pourrait également s'avérer fort intéressant dans le cadre des travaux sur le contrôle des systèmes multiagents qui nécessite d'instrumenter le code des agents en fonction des évènements du système.
- Il pourrait être judicieux de mettre en place un mécanisme permettant de contrôler certaines variables du système afin de lui conférer une plus grande flexibilité. Ainsi, il devrait être possible de modifier certains paramètres propre à chaque agent comme par exemple la taille de ses listes circulaires de messages, le nombre maximal de messages traités par étape élémentaire ou encore de pouvoir définir une extension limitée des vagues de messages de recherche. Il serait envisageable d'associer un compteur aux messages de recherche, ce dernier serait décrémenté à chaque fois que le message passe par une nouvelle accointance. Cela permettrait de désactiver complètement la recherche par vagues ou au moins d'en limiter l'ampleur lorsque les ressources sont très limités. Ces variables seraient internes à chaque agent et seraient héritées par les agents descendants. Il serait dès lors très simple de modifier les paramètres de tout le système (en les fixant pour l'agent initiateur du SMA) ou encore de chaque agent pris indépendamment. Evidemment, des valeurs par défaut permettront au programmeur de ne pas s'en préoccuper dans un cas général d'utilisation.

PrologDoc

PrologDoc pourrait également faire l'objet d'améliorations diverses :

- documentation générée au format XML qui permettrait ensuite de générer des documents dans tous les formats (HTML, pdf...)
- amélioration de l'impression de la documentation, il serait notamment intéressant de pouvoir en imprimer toutes les pages en une opération
- amélioration de la syntaxe des commentaires

1.3.2 De la pertinence de ma formation pour ce stage

La formation que j'ai suivie au sein de l'école me semble avoir été bien adaptée pour cette expérience. Le stage m'a ainsi permis de mettre à profit les compétences qui avaient été acquises en terme de gestion de projet (CVS, ...), de travail d'équipe ou d'autonomie.

Les connaissances en IA qui ont été enseignées durant le cursus de SCIA ont également pu être utilisées à bon escient.

La formation de l'école développe, en outre, notre capacité à apprendre par nous même et de manière efficace, de nouvelles technologies et en ce sens elle a été très bénéfique.

1.3.3 Et maintenant?

Nous avons accepté avec François Chazal, l'opportunité qui nous a été offerte de prolonger le stage afin de participer à l'élaboration d'un prototype qui nous permettra de mettre en oeuvre les développements de nos stages dans le cadre d'une application concrète.

Quant aux perspectives d'avenir, j'espère pouvoir approfondir mon apprentissage en IA dans le cadre d'un master recherche.

Chapitre 5
Bibliographie

5.1 Bibliographie Prolog

[1] La Programmation Logique et langage Prolog
P. NUGUES
Notes de cours 2000

Un cours succint sur la programmation logique (très orienté Prolog) qui permet de prendre en main facilement et efficacement ce langage.

---

[2] Programmer en Prolog
W.F. CLOCKSIN, C.S. MELLISH
Editions Eyrolles, 1985

Un autre cours de Prolog qui a été moins utilisé que le cours de P. Nugues.

---

[3] The Art of Prolog
L. STERLING, E. SHAPIRO
MIT Press, 1986

Il s'agit d'un ouvrage très complet qui est une référence dans le domaine. Peu didactique dans le cadre d'une initiation à Prolog il devient intéressant de le consulter sur des points précis nécessitant un usage pointu du langage

---

[4] SICStus Prolog User's Manual
INTELLIGENT SYSTEMS LABORATORY
Swedish Institute of Computer Science, 2004

Le manuel de référence de l'implémentation SICStus Prolog. Bien qu'assez volumineux (près de 1000 pages) cette documentation reste bien souvent trop allusive, elle a malgré tout été d'une aide précieuse durant le stage pour se renseigner sur les potentialités de SICStus Prolog et sur les modes d'utilisation des différents prédicats offerts au programmeur.

5.2 Bibliographie des systèmes multiagents

[5] Rapport d'activité 2003-2004

DIRECTION TECHNIQUE ISA
Document interne, 2005

Il s'agit du rapport d'activité 2003-2004 du service dans lequel s'est effectué le stage. Il a été utile afin de se faire une vue d'ensemble des activités de l'équipe et de ses domaines actuels de recherche.

---

[6] 18 ans d'intelligence artificielle, ... et maintenant?
PATRICK TAILLIBERT
1999

Cet article décrit, depuis leur origine, les activités en *Intelligence Artificielle* menées durant la carrière de M. Patrick Taillibert. Il donne un point de vue concernant le développement de l'IA dans un contexte industriel.

---

[7] Multiagent systems for Decision support systems, a point of view
PATRICK TAILLIBERT
2004

Lecture indispensable pour bien comprendre le point de vu qu'a l'équipe sur l'usage des SMA.

---

[8] Objet et agents : une étude des structures de représentation et de communications en Intelligence Artificielle
JACQUES FERBER
1989

Thèse de J. Ferber qui a servi de base à de nombreux travaux sur les SMA. Sa lecture en est donc forcément édifiante.

---

[9] Les systèmes multi-agents, vers une intelligence collective
JACQUES FERBER
1995

Livre de J. Ferber fréquemment cité dans le domaine des SMA. Il constitue, là encore, une lecture incontournable sur ce thème.

---

[10] Towards a distributed, environment-centered agent framework
J.R. GRAHAM, K. S. DECKER
2000

Cet article aborde l'organisation interne de la plate-forme DECAF.

---

[11] Programmation orientée-agent : évaluation comparative d'outils et environnements
TONY GARNEAU & SYLVAIN DELISLE
2002

Etude comparative des différents outils et environnements de développement des SMA.

---

[12] Systèmes multiagents : Principes généraux et applications
B. CHAIB-DRAA, I. JARRAS, B. MOULIN

2001

Article présentant un état de l'art intéressant sur les SMA. Il en décrit les principes, en reprend l'évolution historique et aborde leurs applications industrielles.

---

[13] Trust in Multi-Agent Systems, is cryptography a way to achieve trust?
PER JONSSON
Blekinge Institue of Technology

Article intéressant sur l'usage de la cryptographie dans les SMA. Il s'agit d'une extension envisageable qui pourrait être implémentée aisément sous forme de module externe.

Chapitre 6
Glossaire

Accointances : ensemble des a-agents connus d'un a-agent A donné. A peut communiquer par messages avec ses accointances, c'est ALBA qui se charge de toutes les opérations nécessaires aux communications, y compris, si cela est nécessaire, de la localisation et de la connexion avec l'a-agent à contacter.

Agent : au sens de Ferber 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 multiagent, 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.

Agent ALBA ou a-agent : c'est l'entité élémentaire que l'on peut produire en utilisant le niveau d'abstraction fourni par ALBA. C'est un agent au sens courant du mot mais son langage de commande est de bas niveau (échange de termes Prolog) et ne possède pas de sémantique. De même aucune structure d'agent particulière n'est imposée à un a-agent. Lorsque le contexte ne présentera pas d'ambiguïté nous utiliserons le terme « agent » pour « a-agent ». Un a-agent est identifié par un nom unique.

Autonomie : Un agent A est autonome par rapport à B si B ne peut pas prédire à coup sûr le comportement de A.

ALBA : c'est un ensemble de fonctionnalités permettant le développement et le déploiement d'un système multiagent.

BDI : *Belief, Desire, Intention.*Modèle d'agent basé sur ses croyances, ses désirs et ses intentions.

CVS : Concurrent Version System. Solution logicielle de gestion de fichiers qui peuvent être du code source mais aussi n'importe quel type de fichier. Il gère les versions successives d'une même arborescence et garde trace de toutes les modifications opérées sur un dépôt de référence qui contient l'ensemble des données d'un ou plusieurs projets. CVS est tout naturellement adapté au développement logiciel en équipe.

Démon : *Application tournant en tâche de fond de manière continue.*

IA : Intelligence artificielle.

IAD : Intelligence Artificielle Distribuée.

Machine : caractérise le support physique sur lequel s'exécute un a-agent ou sur lequel réside un proto-agent. Une machine est identifiée par son adresse IP.

Message : dans notre cadre terme Prolog valide échangé entre deux a-agents.

Prédicat : expression logique dont la valeur peut être vraie ou fausse selon la valeur des arguments.

Processus : c'est l'entité de même nom manipulée par le système d'exploitation sur lequel s'exécute ALBA (Windows, Unix, ...).

Processus Prolog : c'est un processus créé par l'activation d'un exécutif Prolog. Les processus Prolog communiquent entre eux par sockets TCP/IP dont une implémentation est offerte par SICStus Prolog.

Prolog : langage de programmation logique.

Proto-agent : c'est l'ensemble des informations nécessaires à la création d'un a-agent (code Prolog, exécutable, mais également tous les fichiers de données nécessaires à

l'a-agent...). Plusieurs a-agent peuvent être créés à partir du même proto-agent. A ce titre on peut rapprocher le concept de proto-agent de celui de classe des langages orientés objets, un a-agent étant une instance d'un proto-agent.

SMA : Systèmes multiagents.

Dans notre cadre l'ensemble des a-agents, des proto-agents et des machines sur lesquelles ils résident.

Index

T

É