Université de Caen
UFR de sciences
Laboratoire d’informatique

THÈSE

présentée pour l’obtention du

Doctorat de l’Université de Caen

spécialité : informatique
(arrêté du 30 mars 1992)

par

Bruno DILLENSEGER

Une approche multi-agents des systèmes de bureautique communicante

soutenue le 29 novembre 1996
devant le jury composé de :

Patrice BOIZUMAULT	Professeur à l’Ecole des Mines de Nantes	*rapporteur*
François BOURDON	chercheur au SEPT (CNET Caen)	
Yves DEMAZEAU	chargé de recherche au CNRS (IMAG)	*rapport complémentaire*
Patrice ENJALBERT	Professeur à l’Université de Caen	*directeur*
Marinette REVENU	Professeur à l’ISMRa (ENSI de Caen)	
Michel RIVEILL	Professeur à l’Université de Savoie	*rapporteur*
Marc SINOU	Directeur du SEPT	

“L’image la plus poignante de la solitude humaine ?

Ce pourrait être un homme qui joue de son instrument, chez lui, seul...”

Michel Contat “Ô Solitude” (Télérama no 2402)

Dédié à mes parents et mes frères...

Remerciements

Je remercie en premier lieu M. Marc Sinou, qui m'a permis de réaliser cette thèse au sein du Service d'Etudes communes de La Poste et de France Télécom, où j'ai bénéficié de conditions idéales tant du point de vue humain que logistique. Je lui sais gré également de sa participation au jury.

Je remercie Philippe Maurice et Jean-Marc Deshayes pour m'avoir accueilli avec confiance et soutien, respectivement dans leur groupement (Services de Courrier Electronique) et département (Applications Réparties de Courrier électronique).

Je remercie également l'encadrement administratif du Service de l'Action Scientifique du CNET et de l'ADER PACA, avec qui les contacts se sont toujours agréablement déroulés.

Je remercie les rapporteurs de ces travaux, M. Michel Riveill et M. Patrice Boizumault, qui me font l'honneur de leur évaluation, tout comme M. Yves Demazeau qui a accepté de réaliser un rapport complémentaire.

Je remercie Mme Marinette Revenu, dont j'ai suivi l'enseignement à l'ISMRa, et qui a accepté de participer au jury de thèse.

J'exprime toute ma gratitude à tous les collègues qui ont contribué de façon significative au bon déroulement de cette thèse. L'entourage de compétences du département ARC — puis ACS — m'a bien souvent écouté, conseillé, informé et dépanné. A cet égard, j'exprime ma reconnaissance de façon non ordonnée et non exhaustive à Pierre Touzeau, François Merciol, Michel Milhau, Eric Pillevesse, Pascal Bar.

Je salue le savoir et le talent de Christophe "Tof" Trompette, le concepteur du premier interpréteur PUMA, avec qui la collaboration s'est toujours révélée fructueuse.

Je souhaite à tous les thésards de bénéficier de l'encadrement d'un directeur tel que Patrice Enjalbert. Son expérience, sa disponibilité, son ouverture, son soutien sans faille m'ont permis de mener cette "aventure" à son terme en confiance.

Des remerciements seraient insuffisants pour exprimer le rôle déterminant de François Bourdon au cours de ces quatre dernières années. Cette thèse n'existerait pas sans lui, source

intarissable d'idées, d'enthousiasme et d'énergie, qui m'a apporté bien plus qu'un "encadrement technique".

Ces années de thèse ont d'abord rimé avec solitude puis avec batterie et jazz. Je remercie l'association Caen Jazz Action, en particulier Raynald Fleury, Jean-Benoît Culot et Dominique Voquer, pour m'avoir accueilli dans les "ateliers", ainsi que tous les autres musiciens avec qui j'ai eu la chance de faire le "bœuf", tels Renaud, Deborah, Stéphane… Cela a été rendu possible par la pédagogie de Dominique le Pesant, professeur de batterie et de percussions ; louées soient son ouverture et l'efficacité de sa méthode.

Cette thèse est dédiée à mes parents et à mes frères, dont le soutien et l'exemple ont fait de mes études un fleuve tranquille.

Table des matières

Chapitre 1
Introduction 19

1.1 Problématique **19**

1.2 Présentation du document **20**

Partie 1
Bureautique Communicante :
des Systèmes Répartis Orientés Objets aux Systèmes Multiagents

Chapitre 2
La bureautique communicante 25

2.1 Evolution du système d'information de l'entreprise **25**

Généralités - 25
Problématique actuelle - 26
Propositions - 27

2.2 La bureautique communicante **28**

Introduction - 28
Des applications encore limitées - 29
Les voies de recherche - 30

2.3 Le "Workflow" et les formalismes **31**

Introduction - 31
Opportunité d'un formalisme - 31
Aperçu de quelques approches ([BAS 92a]) - 31
Les "Information Control Nets" - 33

2.4 Le projet CIDRE **34**

Présentation - 34
Particularités de CIDRE - 35
Une application répartie orientée objets - 36
Les limites - 37

2.5 Conclusion **39**

Chapitre 3
Les systèmes répartis à objets. 41

3.1 Le besoin d'objets et de répartition **41**

Caractéristiques de l'approche objet - 41
L'informatique répartie - 43
Les objets au secours de la répartition - 44

3.2 Les premiers Systèmes Répartis à Objets **44**

Introduction - 44
La plate-forme ANSA - 44
Le système GUIDE - 45
Chorus Object Oriented Layer v1 - 46

3.3 La norme ODP **46**

Généralités - 46
Le langage d'entreprise - 48
Le langage d'information - 48
Le langage de traitement - 49
Le langage d'ingénierie - 49
Les fonctions ODP - 51
Les "transparences" d'ODP - 52

3.4 L'architecture CORBA de l'OMG **54**

Présentation générale - 54
Le guide OMA - 55
Spécification de CORBA - 56

3.5 Le système COOL **60**

Le micro-noyau Chorus - 60
COOL v1, une couche orientée objets sur Chorus - 62

La mise en œuvre de COOL v1 - 63
COOL v2 : la prise en compte des standards - 67

3.6 Conclusion **68**

Chapitre 4
L'approche "multi-agents" 71

4.1 Introduction **71**

4.2 L'Intelligence Artificielle Distribuée et le concept d'agent **72**

De l'Intelligence Artificielle à l'IAD- 72
Les systèmes multi-agents- 74
La communication - 75
Structure des agents- 78

4.3 Vers un modèle d'implémentation d'agents **81**

Une programmation "orientée agents" ? - 81
Le modèle acteur - 82

4.4 Les agents vus par les informaticiens et les "communicants" **83**

Introduction - 83
Les agents répartis - 84
Les agents mandataires - 84
Les agents nomades- 85
Les agents intelligents- 86

4.5 Quel enjeu pour les Systèmes Répartis à Objets ? **86**

Introduction - 86
Quelques plates-formes multi-agents- 87
L'offre des SRO - 89
Bilan - 90

4.6 Conclusion **91**

Chapitre 5
La vision multi-agents du système d'information 93

5.1 Une modélisation multi-agents **93**

De l'interopérabilité à la coopération- 93
Obtenir un système uniforme - 94
Optimiser la coopération - 96

5.2 Discussion **98**

L'approche agents - 98

A propos des difficultés de coopération - 99

5.3 Vers des services "intelligents" 100

CIDRE - 100
Messagerie intelligente - 101

5.4 Bilan 102

Partie 2
Outils pour la Modélisation Multi-Agents et Mise en Œuvre

Introduction 105

Chapitre 6
PUMA, un environnement d'implémentation pour l'IAD 107

6.1 La classe de base PUMA 107

Présentation de l'objet Puma - 107
Le dialecte Prolog enrichi - 108
L'interface de contrôle C++ - 111
Interopérabilité Prolog-C++ - 113
Discussion - 115

6.2 Les classes dérivées 116

Arbre d'héritage - 116
La classe interp - 116
La classe agent - 117
La classe objetSMA- 118

6.3 Commentaires 119

PUMA et les agents - 119
PUMA et les acteurs - 120
Exemples- 120

6.4 Quelques détails pratiques de l'intégration 123

Encapsulation de C-Prolog - 123
Intégration à COOL- 125
Les difficultés rencontrées- 125
Le coût des objets PUMA - 127

6.5 Conclusion 127

Chapitre 7
Exemples de mise en œuvre 129

7.1 Premier pas : Workflow multi-agents 129

Présentation - 129
Les agents - 130
Bilan - 131

7.2 Principe d'un système de réservation de salles multi-agents **132**

Présentation générale - 132
Les agents et les services - 133
Négociation et résolution de contraintes - 135
Structure et protocoles de coopération - 138

7.3 Implémentation du système de réservation de salle **138**

Conception des entités - 138
Prolog et la programmation déclarative - 142
Exemple d'exécution - 143

7.4 Conclusion **147**

Chapitre 8
Structures et protocoles de coopération 149

8.1 Introduction **149**

8.2 Le protocole du "groupe d'artisans" **150**

Objectifs - 150
Principe - 151
La gestion interne du groupe - 151
L'invocation du groupe - 153
Evaluation - 154

8.3 Réflexions **156**

Portée de la notion de groupe - 156
Influence du "type" de caractéristique et du "type" de service - - - - - - - - - - - - 158
La question des réseaux à grande échelle - 159

8.4 De la structure de groupe à l'annuaire intelligent **159**

Limites de la structure de groupe présentée - 159
X.500 et la notion d'annuaire - 160
Un groupe d'agents "annuaire intelligent" - 162

8.5 Conclusion **163**

Chapitre 9
Conclusion 165

9.1 Bilan **165**

9.2 Perspectives **167**

Bibliographie

Références bibliographiques. 173

Annexes

Annexe A : documentation PUMA 183
Annexe B : “workflow” multi-agents. 195
Annexe C : la classe interface. 207
Annexe D : la résolution de contraintes 211

Liste des figures

La bureautique communicante

figure 1 : éléments du formalisme de type ICN dans CIDRE 33
figure 2 : exemple d'ICN 34

Les systèmes répartis à objets

figure 3 : les entités structurant un nœud ODP 50
figure 4 : structure d'un canal de communication dans ODP (liaison entre deux objets) . 51
figure 5 : historique des travaux de l'OMG 54
figure 6 : le modèle OMA 55
figure 7 : structure des interfaces d'un ORB 58
figure 8 : structure du noyau Chorus 61
figure 9 : modèle d'objet COOL v1 63
figure 10 : COOL v1 — structure d'un contexte et d'un objet 64
tableau 11 : résumé des principales méthodes de la classe COOL 66

L'approche "multi-agents"

figure 12 : le problème de la synchronisation par message 77
figure 13 : les agents, entre IA classique et connexionisme 81

figure 14 : Le RPC et les deux modes du “Remote Programming” 85

La vision multi-agents du système d’information

figure 15 : représentation uniforme du système 95
figure 16 : exemple de niveaux sémantiques pour un service d’impression 97

PUMA, un environnement d’implémentation pour l’IAD

figure 17 : invocations Prolog -> COOL/C++. 109
tableau 18 : les principaux prédicats “COOL” de Puma 110
tableau 19 : les principales méthodes de l’interface des objets PUMA. 111
figure 20 : invocation C++ -> Prolog 112
figure 21 : les états visibles du noyau Prolog 113
figure 22 : interopérabilité Prolog <-> C++. 114
figure 23 : exemple de traitement de interrupt/1 114
figure 24 : les classes PUMA standard 116
figure 25 : définition de l’activité “générique” d’un objet agent. 117
figure 26 : exemple de calcul parallèle d’une somme par des objets PUMA 122
figure 27 : calcul récursif parallèle de factoriel 123
figure 29 : Comportement d’interpréteur Prolog standard de l’automate PUMA 124
tableau 28 : fonction pEvent(), états et événements 124
figure 30 : myname/1 - détails de l’invocation d’un prédicat PUMA 126

Exemples de mise en œuvre

figure 31 : structure d’un agent de ressource. 134
figure 32 : agents et services du système de réservation de salle 135
figure 33 : schéma d’étape élémentaire de l’activité d’un agent 139

Structures et protocoles de coopération

figure 34 : gestion interne de la structure de groupe 152
figure 35 : invocation de la structure de groupe par une requête complète. 153
figure 36 : invocation de la structure de groupe par une requête incomplète 154
tableau 37 : évaluation et comparaison du réseau de contrat et du protocole de groupe. . . 155
figure 38 : illustration du principe de groupe d’agents annuaires 163

Conclusion

figure 39 : vers une nouvelle génération d'outils pour les systèmes d'information 168

Annexe B : "workflow" multi-agents

figure 40 : exemple fictif d'ICN 195

Chapitre 1

Introduction

1.1 Problématique

Accroître les facultés d'adaptation, supprimer les goulets d'étranglement, améliorer la résolution des problèmes par une distribution ad hoc des tâches à des entités autonomes mais coopérantes, sont autant d'objectifs qui motivent les tendances de notre époque en matière de décentralisation et de communication. Celles-ci se traduisent technologiquement par l'avènement des systèmes informatiques répartis et le développement de canaux de communication aux capacités croissantes. De plus en plus, la conception des systèmes d'information doit prendre en compte la dispersion géographique des données et répartir les traitements pour satisfaire les besoins de coopération entre les ressources (logiciels, matériels, utilisateurs) reliées par le système de communication.

Dans le domaine de la bureautique communicante, les systèmes qui visent à assister la coopération entre les utilisateurs ("Computer-Supported Cooperative Work") ou à favoriser la collaboration entre un utilisateur et l'outil informatique ("Human-Computer Cooperative Work"), se confrontent à des problèmes répartis logiquement et physiquement par nature. La technologie des systèmes répartis alliée aux méthodologies orientées objets fournit un support particulièrement pertinent. Il apporte, en effet, une virtualisation des données, auxquelles on peut alors accéder de façon uniforme quel que soit leur type et leur localisation. Conçues et

implémentées comme des collections d'entités indépendantes en interaction, les applications peuvent prendre en compte les problèmes inhérents à la répartition et à la communication.

Mais la couche dédiée à la gestion de la répartition, de la communication et de l'encapsulation offerte par la technologie répartie à objets reste encore de niveau insuffisant pour les applications finales. Les notions d'autonomie et de coopération impliquent qu'une couche intermédiaire fournisse des modèles et des outils de haut niveau pour conférer aux entités applicatives des capacités de raisonnement, négociation, et adaptation dans un système réparti, ouvert et dynamique. C'est ainsi que les recherches en Intelligence Artificielle Distribuée s'intégrent aux systèmes répartis à objets. Le lien entre ces deux domaines est double : d'une part, toute application répartie orientée objets peut être vue comme un système multi-agents et tirer partie des modèles de coopération, de comportement et de représentation des connaissances développés en IAD. Réciproquement, les systèmes répartis à objets apportent une technologie adaptée à la réalisation pratique, la simulation et l'observation de systèmes multi-agents. Dans cet ordre d'idée, [VER 90] affirme que “l'intelligence au sens prise de décision a besoin des Systèmes Répartis comme les Systèmes Répartis feront appel à l'intelligence”.

1.2 Présentation du document

Les travaux présentés dans ce mémoire visent la mise en relation des besoins en termes d'interopérabilité et d'autonomie des systèmes d'informations avec le support des technologies réparties à objets et de l'Intelligence Artificielle Distribuée. Ce faisant, nous n'entendons pas proposer une étude exhaustive de chacun de ces domaines, mais une approche synthétique guidée par des besoins concrets en bureautique communicante.

La première partie, intitulée “Bureautique Communicante : des Systèmes Répartis Orientés Objets aux Systèmes Multiagents”, propose une présentation de la bureautique communicante, illustrée par des applications existantes dans le domaine du *Workflow*. L'analyse des atouts de l'approche basée sur les systèmes répartis orientés objets montre qu'il s'agit d'une première étape dans la recherche de solutions viables et performantes mais insuffisante en soi. Le bilan des résultats escomptés et obtenus met en lumière des besoins de raisonnement qui nécessitent une modélisation de l'environnement bureautique. Cette démarche nous conduit naturellement dans le domaine de l'Intelligence Artificielle Distribuée

que nous relions à la conception des systèmes d'information et à leur implémentation dans le cadre des systèmes répartis à objets.

La seconde partie a pour titre "Outils pour la Modélisation Multi-Agents et Mise en Œuvre" et se consacre aux aspects pratiques de la thèse. Le cœur de nos outils résulte de l'intégration d'un langage interprété de haut-niveau (Prolog) dans un système réparti orienté objets (Chorus Object Oriented Layer). Prolog pour Univers Multi-Agents est présenté comme un outil d'IAD car il propose un langage d'Intelligence Artificielle étendu pour prendre en compte les problèmes de répartition et de communication. Mais PUMA donne également lieu à une plate-forme d'implémentation de systèmes multi-agents disposant de classes de développement d'agents. La mise en œuvre de nos outils dans une application précise nous a notamment amené à trouver des réponses concrètes aux deux difficultés que pose la coopération entre agents : la recherche efficace d'un agent pour effectuer une tâche et la négociation de cette tâche entre le client et le serveur afin de satisfaire au mieux les besoins du client tout en préservant les intérêts du serveur. Nous décrivons quelques structures de coopérations et protocoles associés susceptibles de faciliter la recherche de l'agent serveur idéal.

Nous concluons par un bilan de ces recherches en termes de résultats vis-à-vis des systèmes et applications répartis à objets, mais aussi en envisageant les perspectives, tant au niveau de l'approche IAD et de la modélisation multi-agents qu'en ce qui concerne l'outil d'implémentation.

PARTIE 1

BUREAUTIQUE COMMUNICANTE :

DES SYSTÈMES RÉPARTIS ORIENTÉS OBJETS

AUX SYSTÈMES MULTIAGENTS

Chapitre 2

La bureautique communicante

Le contexte applicatif de la bureautique communicante constitue le fil conducteur de la thèse. Dans ce chapitre, nous présentons les problématiques liées au besoin d'ouverture et d'adaptation des systèmes d'information en général. Nous en précisons ensuite les manifestations dans le domaine de la bureautique communicante. Enfin, nous détaillons les systèmes de Workflow, dont l'analyse a motivé nos travaux.

2.1 Evolution du système d'information de l'entreprise

2.1.1 Généralités

"La mission principale des services administratifs et des services de gestion est de faire en sorte que la circulation de l'information soit optimale, c'est-à-dire adaptée, rapide, sûre, pratique, économique. La fonction administrative a pour rôle d'approvisionner en données tous les organes concernés par l'activité de l'organisation."

Cette citation, tirée de [LEM 82], expose clairement les besoins à l'origine du développement des systèmes d'information. En manipulant des données électroniques de façon automatisée, l'informatique se doit d'apporter sécurité et rapidité dans le partage, la conservation et le retraitement des informations. Pourtant, les relations intra et inter-entreprises posent le problème d'échange de ces données, tant du point de vue des formats physiques et

logiques, que sur le plan sémantique. Ainsi, le système d'information étant à la fois tourné vers l'intérieur et l'extérieur d'une organisation, ouverture et interopérabilité sont des points clés de son efficacité. Tout le bénéfice tiré d'une informatique et d'une structure organisationnelle maîtrisées localement, ne doit pas se perdre dans des étapes plus ou moins manuelles d'adaptation, de traduction, de réarrangement de l'information qui induisent des goulets d'étranglement, un surcoût notoire, ainsi qu'une perte de retraitabilité, de traçabilité, voire de sémantique.

2.1.2 Problématique actuelle

Jusqu'à une époque récente, le développement des systèmes d'information se focalisait sur la structure organisationnelle de l'entreprise. Le schéma typique d'analyse des besoins consistait alors en une corrélation entre un audit de l'organisation et les cadres théoriques des grands cabinets de conseil [PREVISIA]. L'un des principaux pièges à éviter consiste à calquer directement l'organisation, car l'introduction d'outils informatiques peut (et doit ?) généralement influer sur la structure même de l'organisation [DUP 94]. De plus, il se trouve que l'époque actuelle impose davantage de réactivité aux entreprises, ce qui nécessite une ouverture et une capacité d'adaptation de la part de son système d'information. Par conséquent, les entreprises se recentrent sur leurs processus métiers, qui doivent reposer sur une organisation véritablement souple et dédiée à son soutien efficace. Face à cette nouvelle approche, la diversité et l'hétérogénéité des moyens informatiques disponibles, cumulées à l'incompatibilité des différentes offres propriétaires, compliquent la réalisation d'un système d'information complet, cohérent et opérationnel.

L'autonomie devient de plus en plus un élément fondamental dans l'adaptation des systèmes informatiques à l'évolution imprévisible [THI 93] de l'entreprise, dont l'environnement est géographiquement réparti. Cette évolution imprévisible des comportements réels de l'entreprise, contraint les systèmes informatiques à s'auto-adapter aux changements organisationnels et opérationnels induits.

Dans la période de transition que nous vivons ([DES 93]), suivant le degré de visibilité offert par les applications sur leurs potentialités fonctionnelles, l'utilisateur ne voit plus systématiquement des applications monolithiques mais prend conscience d'objets interagissant. Le potentiel de chaque objet s'exprime en terme de services autonomes, susceptibles d'aider l'utilisateur dans les tâches qu'il doit accomplir.

Devant la profusion de services qui apparaissent, disparaissent et évoluent au sein de l'entreprise, l'utilisateur doit pouvoir être guidé pour trouver les plus pertinents vis-à-vis de ses besoins du moment. Cette assistance nécessite des échanges entre entités informatiques autonomes, qui ne peuvent être réalisées que par l'intermédiaire d'un langage d'interface normalisé susceptible, comme le disent Jean Erceau et Michel Barat à travers leur *Principe intégrateur* [BAR 93], “de déterminer un univers sémantique minimum pouvant être partagé par l'émetteur et le récepteur d'un message et plus généralement par les différents agents du système amenés à communiquer”.

Outre une représentation fonctionnelle des services offerts et/ou demandés, cette recherche doit s'accompagner de possibilités de négociation pour adapter les requêtes aux offres du “marché”, d'apprentissage (mémoire) pour ne pas recommencer systématiquement des recherches inutiles, et de communications sophistiquées offrant des possibilités d'intervention manuelle (intervenant humain) en cas de négociations délicates.

2.1.3 Propositions

De même que le métier devient la référence pour l'organisation de l'entreprise, il convient d'isoler des fonctions génériques et leur mode d'accès au sein du système d'information. A partir de ces briques de base, il devient possible de construire des applications de façon beaucoup plus dynamique. En effet, par le partage de ces boîtes noires entre plusieurs applications, on évite de multiples implémentations de ces fonctions génériques. De plus, l'évolution du système d'information se réduit alors à l'ajout ou au remplacement de briques, ou à la modification de leur assemblage. L'objectif est de disposer d'une architecture pérenne, supportant les changements mais évitant les bouleversements.

De façon immédiate, ce principe de réutilisabilité de fonctions génériques, encapsulées dans des boîtes noires en interaction, tient de l'approche objets. Si l'on prend en considération le caractère hautement réparti de l'entreprise, l'aubaine apportée par les systèmes répartis à objets devient également évidente, car elle permet une interopérabilité à l'échelle du système d'information entier. Enfin, pour s'affranchir de l'hétérogénéité et ne pas s'enfermer dans des solutions propriétaires, les standards tels que ceux définis par l'OMG sont déterminants (voir Chapitre 3).

Toutefois, il s'avère que le support technologique proposé par les systèmes répartis à objets doit s'accompagner de fonctionnalités de haut niveau, en raison de l'ouverture, de la

dynamique et de la répartition à grande échelle des systèmes d'information. En effet, l'activité "tertiaire" (services administratifs ou de gestion d'une organisation quelconque) est par nature orientée vers la résolution de problèmes [STE 86]. La souplesse et la réactivité nécessaires à l'efficacité du système d'information ouvert et réparti exigent des capacités d'autonomie, de coopération, d'adaptation et de raisonnement de la part des entités qui la composent [BOU 96].

Avec une idée précise de rapprochement avec les techniques d'Intelligence Artificielle, [RAS 94] explique que le système d'information de l'entreprise devra intégrer de façon croissante une représentation des connaissances sur les "affaires" (*business*). L'introduction de ces connaissances dans les applications sera de plus en plus cruciale car les entreprises modifient leurs centres d'intérêts et réorganisent leur structure interne. Les applications doivent donc être capables de s'adapter à un nouvel environnement sans devenir obsolète ; pour cela, la combinaison des objets et des règles (au sens de l'Intelligence Artificielle) procurent de la flexibilité. En réalité, [HEW 91] montre qu'il existe une relation étroite entre les systèmes d'informations ouverts et les systèmes d'Intelligence Artificielle Distribuée.

2.2 La bureautique communicante

2.2.1 Introduction

Par bureautique communicante, nous entendons toute application permettant à différentes ressources[1] réparties d'interopérer par le biais d'un système informatique, dans le but de réaliser une tâche bureautique. La bureautique traditionnelle se décline en postes individuels de mise en forme électronique locale de l'information, et de présentation physique non retraitable de celle-ci.

Lorsque plusieurs utilisateurs désirent élaborer en commun un même document, il leur faut trouver un format et un support permettant d'échanger une information la plus retraitable possible. Au cours des étapes plus ou moins manuelles de transfert et de transformation de cette information, toutes les données propres à l'organisation du processus de rédaction coopérative, à la négociation et à la synchronisation entre intervenants, échappe au système. Or, au-delà de la mise en forme individuelle d'informations, l'informatique est susceptible d'apporter une assistance pour structurer et rationaliser le processus de coopération. On parle alors de *groupware* (ou "collecticiel").

1. ressource au sens large ; utilisateur, matériel informatique, logiciel.

Avec le développement des réseaux, on peut déceler un premier niveau de bureautique communicante dans le partage de documents via un système de fichiers réseau ou des échanges par messagerie. Ce principe de travail collectif basé sur le croisement d'outils bureautiques et de communication ouvre la voie à de nombreuses approches généralisées. Les systèmes de CSCW ("Computer Supported Cooperative Work" — travail coopérative assisté par ordinateur) s'attachent à mettre à disposition des moyens informatiques pour assister le travail coopératif entre des personnes géographiquement réparties. L'approche du HCCW ("Human-Computer Cooperative Work") met l'accent sur la coopération entre l'utilisateur et l'ordinateur. Ces deux tendances entraînent des recherches technologiques (e.g. multimedia et réseaux haut débit pour la première, interface "homme-système" pour la seconde), mais également sociologiques et psycho-cognitives.

2.2.2 Des applications encore limitées

Les applications de bureautique communicante actuelles ne rendent que des services limités en comparaison des ambitions. L'outil le plus largement répandu et adopté est bien entendu la messagerie. On peut la considérer comme une application de bureautique communicante, dans la mesure où les interfaces et les formats de message prennent en compte aujourd'hui des données de types variés (documents bureautiques, multimedia). La messagerie peut être également considérée comme une couche de transport asynchrone sur laquelle on peut greffer des outils bureautiques. Ainsi, des applications de rédaction coopérative de documents et de Workflow peuvent être implémentées sur une API de messagerie comme le montrent les prototypes [RCOX400] et [CRCX400], réalisés sur la norme X.400.

La rédaction coopérative de documents et le Workflow sont des applications typiques de groupware. La première consiste typiquement en un découpage logique d'un document en parties dont la rédaction est attribuée par le responsable à différents participants. L'exploit technique consiste alors à réaliser une rédaction répartie synchrone, chaque poste affichant en temps réel les modifications apportées par les autres intervenants. Quant aux applications de Workflow, elles cherchent à automatiser des procédures de circulation et de recueil d'informations (voir 2.3).

En dehors des applications basiques de type messagerie, les applications de bureautique communicante relèvent encore davantage de prototypes ou de maquettes de recherche que de produits opérationnels en condition réelle. Ainsi, les produits de Workflow disponibles ne sont guère convaincants dès que l'on sort des cas d'école. Leur utilisation se confronte soit à un

manque de souplesse, soit à un excès, suivant qu'ils adoptent ou non un formalisme strict de description des procédures. De même, l'utilisation d'applications de rédaction coopérative paraît encore trop lourde et insuffisamment transparente en comparaison des services rendus.

A titre d'exemple, Lotus Notes — un des produits les plus aboutis du moment dans le domaine du groupware - se montre très séduisant du point de vue GED (Gestion Electronique de Documents) par sa puissante notion de base de documents électroniques composites partageable. Mais les mécanismes nécessaires à la mise en place d'une rédaction coopérative, voire d'un Workflow, sont hors de portée de l'utilisateur lambda. Même pour le programmeur, la tâche est assez lourde, et le résultat se révèle peu transparent ou décevant du point de vue fonctionnel.

2.2.3 Les voies de recherche

Il s'avère que les recherches dans le domaine purement technique (e.g. multimedia, réseaux) sont les plus rapides et les plus spectaculaires mais ne créent pas toujours le déclic pour qu'une application deviennent véritablement opérationnelle[2]. Intrinsèquement liée au système d'information, la bureautique communicante hérite des mêmes problématiques et des mêmes solutions. Du point de vue technologique, les systèmes répartis à objets sont d'un grand intérêt pour faciliter la communication entre des entités coopérantes. En particulier, les applications et les données sont vues comme des collections d'objets en interaction, capables de partager des fonctionnalités et d'échanger des informations de façon transversale. En dehors du plan technologique, il existe tout un ensemble d'études d'ordre sociologique, cognitif, ergonomique, dont l'objectif est de contribuer à la définition de modèles d'organisation, de coopération et d'interaction pour le travail collectif.

De façon naturelle, ces problématiques de coopération et d'activité collective offrent un champ d'investigation et d'application dans le domaine des Systèmes Multi-Agents [MOU 93]. Des techniques d'Intelligence Artificielle Distribuée peuvent alors être adoptées pour aborder certaines problématiques sous un angle de représentation de connaissances et de résolution de problème distribuée. Comme le souligne [BOW 90], en préface d'un recueil d'articles consacrés au CSCW, les approches multi-agents se multiplient. L'enjeu est de fournir

2. Notons toutefois que le développement d'Internet et le nouveau concept d'"intranet" ouvre aujourd'hui de nouvelles perspectives pour aborder ces problèmes de coopération, notamment au travers d'interfaces utilisateur simples, conviviales et standard (cf. "browser web").

une assistance “intelligente” aux utilisateurs, voire aux applications, par le biais d’agents autonomes et coopérants.

2.3 Le “Workflow” et les formalismes

2.3.1 Introduction

Parmi les applications de bureautique communicante, celles de type *Workflow* sont plus particulièrement à l’origine de nos travaux. Par conséquent, la connaissance des problématiques liées à la mise en œuvre réelle de telles applications est importante pour comprendre la justification de notre approche. Nous les exposons ici, puis nous explicitons leur complexité en 2.4, au travers de l’analyse approfondie de l’application CIDRE développée au SEPT.

2.3.2 Opportunité d’un formalisme

Le choix d’un formalisme est un point critique pour une application de type Workflow. Il s’agit de modéliser une procédure en essayant de garantir à la fois rigueur et souplesse. Certaines réalisations laissent les utilisateurs entièrement libres en considérant que, pour toute procédure, chaque intervenant sait ce qu’il doit faire et à qui il doit transmettre le document. A l’inverse, d’autres appliquent une procédure prédéterminée.

L’unique intérêt des premières est de remplacer le papier par des documents électroniques car elles laissent la connaissance sur les procédures bureautiques complètement répartie entre les différents utilisateurs. Par contre, la structuration des secondes fournit un cadre de raisonnement (avant, pendant et après exécution), et autorise la simulation des procédures ainsi que leur automatisation contrôlée. L’inconvénient est, bien sûr, le manque de souplesse, surtout lorsque quelque chose d’inhabituel survient. De plus, l’administrateur doit connaître parfaitement les procédures pour pouvoir décrire les schémas de circulation.

2.3.3 Aperçu de quelques approches ([BAS 92a])

<u>“Logical Routing”</u>

Il s’agit d’une modélisation de la circulation de documents vus comme des entités intelligentes et autonomes capables de choisir l’étape à suivre. Les chemins possibles sont définis dans la spécification de routage qui se compose d’un ensemble de sites correspondant aux étapes envisagées.

La migration d'un site à un autre peut être décidée en fonction de la valeur d'un champ du document, de l'interrogation d'une base de données, de considérations temporelles ou suivant d'autres critères concernant l'état du système. Lorsque le document arrive sur un site de terminaison (fin de la circulation automatique), le routage peut se poursuivre manuellement. Il est également possible d'envoyer un document vers plusieurs sites en même temps. Dans ce cas, les étapes concurrentes ne peuvent décider individuellement de la migration suivante et le routage ne peut reprendre que lorsqu'elles sont terminées.

Ce formalisme ne prend en compte que les questions de routage et il n'existe aucune modélisation concernant les traitements effectués sur les documents lors des étapes.

Les “Objets Coopératifs”

Etudié à l'Université Toulouse I, ce formalisme se présente sous la forme d'un langage de conception orienté objet dédié à la modélisation du comportement et de la coopération des objets dans les systèmes concurrents ([BAS 92b]). Son objectif est de combiner une approche objet poussée avec les concepts des réseaux de Petri en définissant des *structures de contrôle d'objet* (ObCS) et *structures de contrôle d'opération* (OpCS) basées sur des réseaux de Petri objets.

Le projet ITHACA[3] [JAM 90], en utilisant une modélisation par des réseaux de Petri annotés en langue anglaise simplifiée (“Playscripts”), s'est appuyé sur un fondement théorique similaire.

“Electronic Circulation Folders”

Ce modèle a été développé pour le projet ESPRIT ProMInanD[4] [KAR 90] qui a rejeté le principe des réseaux de Petri ou dérivés en estimant qu'il amenuise la flexibilité et complique le traitement d'exception.

L'ECF (dossier électronique de circulation) réalise la circulation de documents en suivant de façon très souple une séquence d'étapes définie par une spécification de migration. Chaque étape est caractérisée par un programme (“step program”) à invoquer ainsi que par les documents et l'intervenant concernés. Un ECF peut lancer un nouvel ECF en parallèle ou se synchroniser avec un autre ECF.

3. “Integrated Toolkit for Highly Advanced Computer Applications”
4. “Extended Office Process Migration with Interactive Panel Displays”

Les ECF ne possédant aucun modèle de structure de données des documents qu'ils manipulent, toutes les étapes et toutes les décisions (choix entre plusieurs chemins possibles) nécessitent l'écriture de progiciels spécifiques par un programmeur.

2.3.4 Les "Information Control Nets"

Les ICN ont été spécifiés au Xerox PARC pour modéliser des flux d'information dans une administration ([ELL 79]). Dérivé des réseaux de Petri, ce formalisme en possède les propriétés tout en simplifiant la représentation graphique. De plus, il laisse apparaître explicitement les productions et consommation d'éléments d'information. Les ICN ont été souvent adoptés dans les applications de Workflow, telles que CIDRE (voir 2.4) ou l'offre commerciale de BULL.

Afin d'illustrer la notion de formalisme de Workflow, et pour éclairer la partie 2.4, la figure 1 présente le formalisme résultant de la transposition immédiate des ICN dans CIDRE. L'ICN y est utilisé pour décrire la circulation (i.e. l'enchaînement des tâches) d'un document ou d'un dossier électronique. Un ICN est un treillis dont les nœuds sont des actions, des connecteurs logiques, ou des macros (sous-ICN).

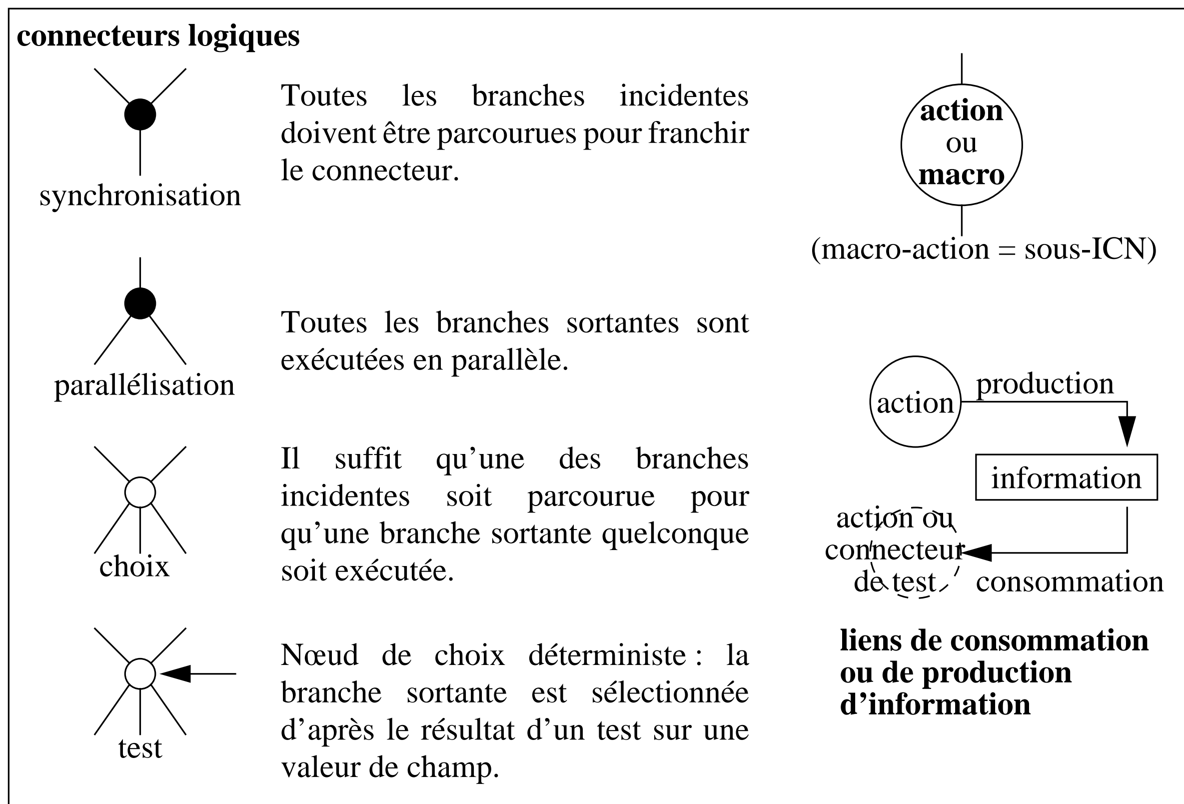


figure 1 : éléments du formalisme de type ICN dans CIDRE

Ainsi, l'ICN de la figure 2 met en jeu des actions (a1, a2, …a10), des connecteurs logiques (et/ou) ainsi qu'un champ c1 représentant un élément d'information du document. Le schéma se lit de haut en bas et indique en particulier que a1 est la première action et a10 la dernière. Les relations fléchées représentent une production ou une consommation d'information par une action. Par exemple, a1 produit l'information c1 qui est consommée par a10. Les autres relations matérialisent les précédences entre actions, de haut en bas.

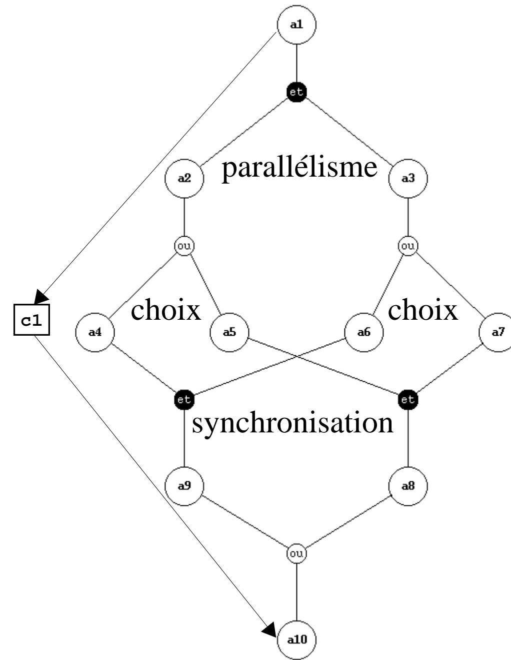


figure 2 : exemple d'ICN

2.4 Le projet CIDRE

2.4.1 Présentation

<u>Généralités</u>

Le système CIDRE [BOU 89] (Circulation Intelligente de Dossiers REpartis) est une application de type Workflow. Sa fonction est de substituer les nombreux documents-papier qui cheminent de bureau en bureau par des documents électroniques parcourant un réseau de postes de travail informatiques. Ce parcours est spécifié par un schéma de circulation strict, prenant en compte la structure des documents et une évolution de droits d'accès.

Un des objectifs spécifiques de l'approche CIDRE est de répondre à la répartition naturelle de la circulation de documents par une répartition physique et logique effective de l'application. Les documents ne sont pas des données gérées par un logiciel, mais des entités autonomes et nomades. Basé sur un formalisme de schéma de circulation, le système global doit apporter des fonctionnalités de simulation, de contrôle et de suivi. Enfin, des mécanismes "intelligents" sont censés guider la circulation.

Le premier niveau de solution technique promu par CIDRE, mais aussi les nombreuses difficultés rencontrées et l'insuffisance des capacités de raisonnement, ont constitué le point de départ de cette thèse ([DIL 92]).

Les abstractions

Un document CIDRE se compose de trois parties : le formulaire, le schéma de circulation, et l'historique de la circulation. Le formulaire est structuré en *atomes logiques* qui regroupent un certain nombre de *champs*, unités élémentaires d'information. Une procédure correspond à un ensemble de documents structurés qui composent un dossier. L'*administrateur CIDRE* est chargé de créer des dossiers génériques qui seront ensuite instanciés pour un cas précis par les utilisateurs. L'utilisateur qui crée une instance particulière de dossier est qualifié d'*initiateur.*

Le schéma de circulation spécifie un certain nombre d'*actions* qui seront effectuées par des *intervenants*, représentés par des *rôles*. A priori, elles consistent à remplir des champs dans un document, mais il est possible d'en définir d'autres. Le *visa* est un champ particulier par lequel un intervenant endosse la responsabilité de ce qui figure sur certains autres champs s'y rapportant.

2.4.2 Particularités de CIDRE

L'"intelligence"

Comme son nom l'indique, il s'agit d'introduire des capacités de raisonnement dans le système. A l'heure actuelle, l'intelligence réside dans un système expert qui sert à générer les schémas de circulation ([BES 88]). En effet, l'administrateur doit définir un schéma de circulation pour chaque procédure qu'il souhaite implémenter. Pour cela, il rédige une base de faits décrivant les actions, les productions et consommations d'information, ainsi que les précédences devant être respectées. Le système expert corrige les incohérences et élimine les ambiguïtés en demandant des précisions à l'administrateur.

Des formalismes à base de réseaux de Petri

Le formalisme choisi pour les schémas de circulation est celui des "Information Control Nets" ou, plus récemment, celui des objets coopératifs. Ces deux formalismes sont dérivés des réseaux de Petri. Le premier est adapté à une visualisation simple et lisible d'une procédure par un utilisateur. Le second, basé sur des réseaux de Petri à objets, se destine plutôt à une représentation interne car il offre un pouvoir d'expression beaucoup plus grand.

L'adoption de formalismes dérivés des réseaux de Petri permet de bénéficier d'un certain nombre d'outils génériques (conception et simulation) et de propriétés intéressantes. En

particulier, la modularité permet d'inclure des sous-procédures dans une procédure globale (les *macros*).

Gestion des droits

CIDRE possède une gestion assez fine des droits d'accès. Ces droits s'appliquent aux atomes logiques et se déclinent en quatre niveaux :

- aucun droit — l'atome logique est invisible.
- lecture seule — l'atome logique est visible.
- lecture et écriture — les champs de l'atome logique sont éditables.
- lecture, écriture et validation[5].

Les droits évoluent au cours de la circulation ([PIL 89]). En effet, la validation d'un champ empêche toute modification ultérieure et un visa annule les droits d'écriture et de validation sur les atomes logiques concernés.

Commentaire

En exploitant les ICN, dérivés des réseaux de Petri, le formalisme de CIDRE se rapproche évidemment beaucoup de celui des Objets Coopératifs. D'ailleurs, des études plus approfondies à ce sujet sont en cours. Mais, en dehors des possibilités de parallélisme, de synchronisation et de choix dans la circulation des documents, il existe aussi des points communs entre CIDRE et les autres modélisations :

- les intervenants sont vus au travers de *rôles* auxquels sont associés des utilisateurs au moment de la circulation effective,
- les "sous-tâches" de ProMInanD sont le pendant des macros de CIDRE,
- les documents modélisés par Logical Routing sont considérés comme intelligents et autonomes, ce qui est également l'orientation de CIDRE.

2.4.3 Une application répartie orientée objets

L'analyse des besoins de CIDRE, suite à une première version réalisée sur une technologie C++/RPC, a abouti à la spécification par le SEPT d'une couche de communication orientée objets. Cette couche a été développée par la société Chorus Systèmes au dessus du

5. Pour tout atome logique, il existe un et un seul intervenant qui possède le droit de validation. En l'exerçant, il fige le contenu des champs et en endosse la responsabilité.

micro-noyau Chorus (voir Chapitre 3), en collaboration avec l'INRIA. Elle a permis d'implémenter CIDRE sous forme d'une collection d'objets répartis :

- objets bureautiques (dossiers et documents), qui migrent d'un poste de travail à un autre au gré de la localisation des intervenants ;
- objets serveurs (création de dossiers, annuaire, délégation, suivi des documents en circulation).

Pour favoriser l'intégration et la communication entre ces entités, une couche dite de "Gestionnaire d'Objets" a été développée au dessus de COOL. Les G.O. sont des objets COOL, répartis sur l'ensemble des postes de travail. Chaque G.O. exerce un rôle d'administration des objets résidant sur la même machine. Par le biais de protocoles spécifiques, les G.O. coopèrent pour rendre des services répartis de niveau supérieur à ceux fournis par COOL, en matière de localisation, migration, manipulation des objets bureautiques, et pour l'accès aux objets serveurs.

2.4.4 Les limites

Le manque de souplesse

Le prototype actuel est très rigide car il se contente d'interpréter l'ICN de manière figée et systématique. Ceci s'accorde mal avec l'utilisation du système par des personnes, qui nécessite une réelle souplesse. En effet, un utilisateur doit pouvoir déclencher des exceptions tout en restant dans le cadre de la procédure. Par exemple, il peut retarder ou refuser d'effectuer une action pour des raisons d'emploi du temps, de manque de compétence ou d'information, ou s'il détecte des erreurs dans le contenu du document. Il peut également souhaiter corriger ou annuler une action a posteriori.

Des blocages à éviter

Par ailleurs, CIDRE peut rencontrer des cas d'exception dus à des blocages.

- Il s'avère impossible d'accomplir une action : pas d'utilisateur ad hoc disponible, ou un utilisateur a accepté la requête mais oublie d'effectuer l'action. Ce type de blocage peut parfois être résolu en laissant une certaine souplesse aux utilisateurs, qui négocient entre eux certains aménagements dans la circulation (suppression/ajout/avancement/retard d'une étape, changement d'un intervenant…).

- Le parcours correspond à un chemin bloquant de l'ICN. En effet, un ICN n'est accepté par le générateur que s'il contient au moins un parcours non bloquant, mais cela n'empêche pas l'existence d'un ou plusieurs chemins bloquants[6]. Le document doit donc être capable d'anticiper l'apparition de tels blocages, résultats de choix soit "arbitraires", soit fonction de la valeur de champs.

Permettre des comportements plus évolués

Pour le confort d'utilisation, les documents doivent être capables de proposer une valeur par défaut pour certains champs, et ce de façon adaptée à chaque utilisateur (e.g. nom, service, adresse). De plus, un document doit pouvoir invoquer d'autres ressources telles que des applications (e.g. base de données) ou des matériels (imprimante, fax…).

Par ailleurs, la circulation doit pouvoir être optimale par rapport à des priorités propres à un document, en termes de coût, de délai, de qualité ou de sécurité. En outre, le test déterminant le choix d'une branche doit pouvoir prendre en compte d'autres critères que la valeur de certains champs. Le document doit donc être capable de déterminer, à un instant donné, la meilleure action, la meilleure branche, le meilleur utilisateur. Il doit aussi pouvoir revenir sur ses choix en cas de problème.

Conclusion

Pour l'ensemble de ces raisons, il apparaît que la notion de document autonome et actif est fondamentale mais insuffisante. Le comportement du document doit inclure des capacités de raisonnement sur la procédure elle-même, sur le contexte de son exécution, sur les actions mises en jeu. L'environnement de la circulation présente certains aspects relativement statiques (structure de l'organisation, hiérarchie...), et d'autres typiquement dynamiques (structure d'un projet, présence et emploi du temps des intervenants, délégations, fonctionnement des postes de travail...).

De plus, des capacités d'observation et d'apprentissage permettraient d'améliorer une procédure, qui, à partir d'une certain degré de complexité, est très difficile à formaliser a priori par l'administrateur CIDRE. Enfin, ces mécanismes "intelligents" nécessitent une collecte d'informations qui sont réparties dans tout le système (cf. disponibilité et caractéristiques des utilisateurs et des ressources du système).

6. Par exemple, la figure 2 présente deux parcours non bloquants ({a1, a2, a3, a4, a6, a9, a10} et {a1, a2, a3, a5, a7, a8, a10}) mais aussi deux parcours bloquants ({a1, a2, a3, a4, a7} et {a1, a2, a3, a5, a6}).

2.5 Conclusion

L'efficacité du système d'information est confrontée à un impératif d'ouverture et d'interopérabilité dans un environnement dynamique, réparti et hétérogène. En ce qui concerne les applications de bureautique communicante, le niveau de service rendu est, entre autres, conditionné par la qualité de leur intégration au système d'information. Dans les deux cas, l'approche des systèmes répartis à objets est prometteuse, car elle permet d'isoler des fonctions génériques au sein d'entités indépendantes, réutilisables et combinables.

Mais, pour offrir des fonctionnalités suffisamment avancées et transparentes, ces entités doivent aussi comporter des capacités d'autonomie, de raisonnement et d'adaptation. Pour faire face à des besoins fondamentaux de coopération et de résolution de problème, des modèles et des techniques relevant de l'Intelligence Artificielle Distribuée peuvent s'avérer précieuses. Nous détaillons successivement dans les deux chapitres suivants l'apport des systèmes répartis orientés objets et les caractéristiques de l'approche multi-agents.

Chapitre 3

Les systèmes répartis à objets

Dans le chapitre précédent, nous avons montré la nécessité d'une technologie adaptée à la répartition, l'hétérogénéité et la complexité des problèmes posés par le "groupware" et la bureautique communicante. Puisque nous avons promu l'approche des systèmes répartis à objets dans ce contexte, il convient d'en expliquer les tenants et les aboutissants. Après une présentation des objectifs, du modèle général et des abstractions liés à cette approche, nous proposons donc quelques systèmes, puis nous exposons une norme. Enfin, nous décrivons de façon assez détaillée la couche de communication orientée objets COOL, dont la spécification a pour origine le développement d'applications de "groupware" (cf. Chapitre 2 : rédaction coopérative, CIDRE). Une certaine connaissance de COOL est importante pour la parfaite compréhension de la seconde partie du document, qui met en œuvre une plate-forme multi-agents basée sur ce système.

3.1 Le besoin d'objets et de répartition

3.1.1 Caractéristiques de l'approche objet

Motivations

L'approche objets relève de deux aspirations. D'une part, il s'agit d'un modèle de représentation des connaissances, issu de l'Intelligence Artificielle. D'autre part, elle se traduit

par des techniques de conception et de programmation qui répondent à des principes de génie logiciel tels que modularité, réutilisabilité, maintenabilité. D'une manière générale, la conception orientée objets permet d'obtenir une représentation précise et lisible d'une application à travers ses différents composants. De plus, une telle modélisation rend possible le maquettage rapide et la simulation, notamment à l'aide des outils des Ateliers de Génie Logiciel ("AGL").

Principes et définitions

Rappelons brièvement les principaux fondements et définitions du modèle objet, tels qu'ils sont désormais largement admis. L'objet est une entité informatique qui regroupe des données privées — ses **attributs** — et des opérations — ses **méthodes** — pour les manipuler. Pour y accéder, un attribut ou une méthode doit être désigné par son **sélecteur**. Cette *encapsulation* des données et du comportement fait de l'objet une "boîte noire", que l'on utilise indépendamment de sa constitution interne. De plus, ceci permet de définir des **réflexes**, i.e. des opérations déclenchées de façon automatique par certains événements. Les réflexes les plus courants sont les *constructeurs* et les *destructeurs*, exécutés respectivement lors de la création et de la destruction d'un objet.

L'invocation de l'objet, et en particulier l'accès à ses attributs, ne peut se faire que par envoi de **messages**. L'ensemble des méthodes que l'on peut invoquer sur un objet constitue l'**interface** de cet objet. Un **type** caractérise une catégorie d'objets offrant une interface similaire, i.e. dont les opérations définissent un mode de manipulation commun. La notion de *conformité* d'un type T1 à un type T2 permet d'exprimer la possibilité d'invoquer un objet de type T1 de la même façon qu'un objet du type T2.

En dépit d'une interface similaire, l'implémentation des opérations de deux objets de même type n'est pas nécessairement la même ; il peut d'ailleurs s'agir d'objets de natures tout à fait différentes. La notion de **classe** permet de définir des catégories d'objets semblables du point de vue de leur constitution (attributs, méthodes). Une classe définit donc une réalisation particulière d'un type. Les objets sont alors des *exemplaires* — ou **instances** — de cette classe. L'**héritage** permet de définir une classe par spécialisation d'une autre, en redéfinissant ses méthodes ou en étendant l'interface par de nouvelles opérations. Par le biais de l'héritage multiple, la classification des objets — ou *taxinomie* — engendre un graphe orienté. Ce graphe se limite à un arbre dans le cas de l'héritage simple, i.e. si une classe ne peut hériter que d'une

classe au plus. Quoi qu'il en soit, l'héritage permet de créer un objet **polymorphe**, dans la mesure où il peut être invoqué en tant qu'exemplaire d'une classe parente.

Mise en œuvre

L'encapsulation des données et des comportements, ainsi que l'héritage et le polymorphisme, permettent une modularité poussée et le découpage d'une complexité globale en entités simples dont le rôle est clairement cerné, mais dont l'implémentation particulière est indifférente. Cela facilite la maintenance et contribue à augmenter le niveau de complexité globale gérable dans le développement des applications. Dans les langages orientés objets, les classes et l'héritage fournissent un mécanisme élégant et intégré au modèle de programmation facilitant la réutilisabilité.

3.1.2 L'informatique répartie

Généralités

Les recherches dans le domaine de l'informatique répartie ont plusieurs origines :

- la complexité en temps de calcul et en espace mémoire de certains problèmes imposent un **découpage** des traitements et des stockages sur plusieurs machines autonomes ;
- la robustesse d'un système informatique, sa disponibilité et sa résistance aux fautes impliquent une certaine **redondance** entre des machines distinctes ;
- la répartition physique naturelle des systèmes d'information, mise en évidence par le développement des réseaux, entraîne un besoin d'**indépendance à la localisation** pour la communication et l'accès aux ressources.

Les systèmes répartis ont pour objectif de fournir des mécanismes de base aux applications pour prendre en compte ces besoins de façon **transparente**.

L'approche client-serveur

De façon croissante, la répartition devient incontournable car la conception de systèmes monolithiques répondant aux besoins actuels n'est plus viable du point de vue coût et complexité. C'est ainsi que les architectures applicatives réparties de type client-serveur se développent. Elles consistent à placer les fonctions de présentation et d'interface sur des ordinateurs clients, adaptés aux traitements graphiques, voire multimedias. Le réseau relie ces postes aux serveurs, dotés de fortes capacités de traitement, qui fournissent des ressources applicatives et/ou matérielles partagées.

3.1.3 Les objets au secours de la répartition

Bien que répondant à l'impératif de répartition, l'approche client-serveur a rapidement montré certaines limites. Confrontée à une hétérogénéité des systèmes, elle engendre une importante inflation des connaissances requises pour les programmeurs. Alors que le développeur d'applications centralisées sur *main-frame* bénéficie d'un ensemble réduit et cohérent d'outils, une approche client-serveur basique contraint les développeurs à maîtriser les réseaux et les environnements spécifiques à tous les postes clients et serveurs réseaux. De plus, la réutilisabilité escomptée par une simple approche orientée objets est mise à mal par cette hétérogénéité ([GID 94]). Les spécificités des matériels et des logiciels d'exploitation persistent, et la dépendance vis-à-vis des interfaces graphiques, des réseaux et les langages de programmation compliquent notoirement la portabilité.

Pour s'affranchir de l'hétérogénéité, il faut intégrer le paradigme objet aux systèmes d'exploitation. En effet, les systèmes orientés objets rendent les architectures client-serveur viables car, en plus des atouts propres aux objets, ils apportent une encapsulation cohérente des réseaux et de la répartition, grâce aux notions de messages et d'objets ([STI 94]). La conception orientée objets permet également de s'abstraire des différents langages d'implémentation mis en œuvre.

3.2 Les premiers Systèmes Répartis à Objets

3.2.1 Introduction

Le déroulement de nos travaux coïncide avec l'émergence des premiers systèmes répartis à objets. En reprenant et étendant les modèles en cours d'élaboration dans ODP (voir 3.3), ces implémentations ont contribué à la maturation de la norme. Elles ont accompagné la définition de l'architecture CORBA (voir 3.4) et précédé l'apparition des produits conformes. Aujourd'hui, l'évolution de ces précurseurs les rapprochent des travaux de l'OMG.

3.2.2 La plate-forme ANSA

Le projet ANSA (Advanced Network Systems Architecture), puis le projet Esprit ISA (Integrated System Architecture), ont abouti à la réalisation d'un système visant à assurer un support à la répartition transparent, y compris en milieu hétérogène. La plate-forme du système réparti résulte de l'interaction de noyaux placés sur chaque machine et spécifiques à celles-ci. Les routines de transparence offrent plusieurs options : transparence à la localisation d'un

objet lorsqu'on l'invoque, placement quelconque des objets par le système, et transparence des accès concurrents.

La **capsule** représente l'unité de répartition, qui regroupe un certain nombre d'objets sur un site donné. Pour interagir avec d'autres capsules de façon transparente à la localisation et à l'hétérogénéité, une capsule dispose de *routines souches* ("stub routines"), ou **talons**. Ces routines sont produites pour chaque capsule par un compilateur particulier, à partir d'une description des interfaces réalisée avec un **IDL** ("Interface Definition Language"). L'**exportation** d'une interface par une capsule "serveur" consiste à déclarer auprès d'un **courtier** (cf. "trader" dans [LIN 90]), l'ensemble des opérations offertes aux autres capsules. L'**importation** d'une interface permet à une capsule "cliente" d'obtenir du courtier la référence d'une capsule "serveur" ayant exporté cette interface.

3.2.3 Le système GUIDE

GUIDE est l'Environnement Distribué et Intégré des Universités de Grenoble. Il se compose d'un système d'exploitation réparti développé au dessus d'UNIX, et d'un langage spécifique orienté objets. Le modèle prend en compte les notions d'objet et de classe, mais aussi de type et de conformité de type. L'architecture de la **machine virtuelle** GUIDE repose sur un certain nombre d'abstractions :

- Les **domaines** constituent des espaces d'adressage et d'exécution. Ils peuvent être répartis sur plusieurs sites, et plusieurs domaines peuvent cohabiter sur un même site.
- Ces domaines accueillent les **objets**, qui sont passifs, et les activités qui les animent.
- Avant d'invoquer un objet, l'objet appelant doit **lier** celui-ci dans son domaine, ce qui peut entraîner une extension du domaine.
- La **mémoire virtuelle d'objets**, composée de l'ensemble des **mémoires principales** des sites, accueille physiquement les objets liés dans au moins un domaine, et qui sont donc susceptibles d'être utilisés.
- La **mémoire permanente d'objets** assure la persistance des **objets permanents**.

Un objet peut être liés dans plusieurs domaines s'il est invoqué par des objets de domaines différents. Cependant, il ne se trouve physiquement que sur un seul site à un instant donné,

dans une mémoire principale. Son placement dynamique est assuré de façon transparente par le système suivant deux principes :

(1) Si l'objet invoqué est distant mais n'est pas utilisé, il est migré auprès de l'objet appelant pour permettre une invocation locale.

(2) Si l'objet distant invoqué est déjà utilisé et ne peut donc pas être migré, l'invocation se fait à distance par RPC ("Remote Procedure Call").

La désignation des objets se fait soit de façon indépendante de la localisation via des identificateurs d'objets gérés par le système, soit par des noms symboliques en ce qui concerne les objets permanents.

3.2.4 Chorus Object Oriented Layer v1

Réalisé suivant les spécifications du SEPT, COOLv1 constitue la base d'implémentation de nos travaux. Un de nos objectifs était de mettre en lumière l'intérêt de ce système, tant du point de vue général de l'approche répartie orientée objets que de ses fonctionnalités spécifiques. COOL est décrit en détail en 3.5, depuis ces origines jusqu'à sa version actuelle, (postérieure à nos développements).

3.3 La norme ODP

3.3.1 Généralités

Le besoin d'une norme

Comme l'exprime [NEW 94], le succès de l'informatique répartie à objets dépend de son aptitude à faire collaborer des machines, des services et des applications d'origines diverses. D'une part, les utilisateurs apprécient l'indépendance vis-à-vis de tel ou tel industriel et la possibilité de configurer finement leur système en choisissant librement les modules ad hoc. D'autre part, les industriels cherchent à réduire leurs coûts de développement par une portabilité et une réutilisabilité accrues, tout en séduisant les utilisateurs avec des produits ouverts. Ceci nécessite l'établissement de normes permettant de définir des modes d'interaction indépendants du matériel, du logiciel d'exploitation (machine et réseau), et des environnements de développement (e.g. langage de programmation). [MER 94] explique en détail les travaux en cours dans ce domaine, menés par des organismes de normalisation et des industriels.

L'ODP

Plusieurs organismes de normalisation (ISO, IEC, AFNOR...) travaillent à la définition de normes concernant la spécification des systèmes répartis ouverts, désignés sous le vocable d'ODP (Open Distributed Processing). Développé par l'ISO et ITU-T (anciennement CCITT), le modèle de référence de la norme ODP (RM-ODP) offre à tout type d'application un cadre conceptuel pour définir une architecture répartie favorisant l'interopérabilité et la portabilité, y compris en environnement hétérogène et de façon transparente. RM-ODP ([ODP 95]) comprend 4 volets :

(1) une présentation générale des motivations et des concepts clés de l'architecture (ISO 10746-1 ou ITU-T X.901),
(2) une définition précise des concepts nécessaires à la spécification des traitements répartis (ISO 10746-2 ou ITU-T X.902),
(3) une prescription d'un ensemble de concepts, structures, règles et fonctions caractérisant le traitement réparti ouvert (ISO 10746-3 ou ITU-T X.903),
(4) des techniques de description formelle des concepts du modèle (ISO 10746-4 ou ITU-T X.904).

Les cinq “points de vue”

Afin de permettre le découpage de la spécification d'un système ODP ou de ses composants en parties abordables, homogènes et adaptées aux besoins particuliers des différents membres d'une équipe de développement, RM-ODP définit cinq projections :

- le **point de vue de l'entreprise**, qui définit les besoins, l'organisation et les stratégies propres à celle-ci ;
- le **point de vue de l'information**, qui apporte la sémantique des informations, de leurs relations et de leur utilisation ;
- le **point de vue des traitements**, qui offre une décomposition fonctionnelle indépendante du support informatique ;
- le **point de vue de l'ingénierie**, qui spécifie l'infrastructure nécessaire à la répartition de l'application ;
- le **point de vue de la technologie**, qui se consacre aux technologies utilisées pour l'implémentation.

La spécification de chaque point de vue repose sur un langage propre, qui définit un ensemble de concepts et de règles s'y rapportant (ISO 10746-3 ou ITU-T X.903). De par la nature même d'ODP, qui vise à permettre une description abstraite et indépendante de la technologie, le langage de technologie est très restreint. En revanche, les autres langages sont beaucoup plus riches. Nous les présentons dans ce qui suit.

3.3.2 Le langage d'entreprise

Le point de vue de l'entreprise concerne sa structure et son organisation. Il distingue des *objets*, des *communautés*, ainsi que des *rôles* particuliers au sein d'une communauté, relatifs à des règles de fonctionnement. Les objets peuvent être actifs, tel un membre du personnel ou un client, ou bien passifs. Il s'agit alors d'entités abstraites ou concrètes liées au métier de l'entreprise (un compte bancaire, des capitaux…). Les communautés regroupent un ensemble d'objets qui concourent à la réalisation d'un objectif global, la manière d'y parvenir étant spécifiée dans un *contrat*.

Quant aux rôles, ils obéissent à des *politiques* qui définissent les permissions, les interdictions et les obligations. Dans ce point de vue de spécification, les seules actions qui sont pertinentes dans le modèle sont celles qui modifient les politiques. Enfin, on note que ces politiques ne doivent provenir que de l'organisation elle-même, et en aucune façon de contraintes liées à l'implémentation.

3.3.3 Le langage d'information

Le point de vue de l'information permet de spécifier la sémantique des informations et de leur traitement, par le biais de trois types de *schéma*. La conjonction de ces trois schémas permet de définir le *gabarit* d'un objet d'information :

- un *schéma d'invariant* détermine les états et les changements d'état possibles pour un ou plusieurs objets, en fixant un ensemble de règles devant toujours être vérifiées (les autres types de schémas doivent donc respecter ces contraintes) ;
- un *schéma statique* décrit l'état d'un ou plusieurs objets à un instant donné ;
- un *schéma dynamique* spécifie les changements d'état autorisés pour un ou plusieurs objets.

Un objet d'information peut être atomique, ou résulter de la composition de plusieurs objets, l'état résultant étant alors défini par la combinaison des états de ces objets.

3.3.4 Le langage de traitement

Résolument orienté objet, le langage de traitement sert à spécifier fonctionnellement les applications et les fonctions ODP sous la forme de configurations d'objets de traitement en interaction par le biais d'*interfaces* :

- *interface d'opération*, permettant l'invocation d'un serveur par un client qui fournit certaines informations en vue d'exécuter une fonction ;
- *interface de flot*, supportant des séquences d'interactions, voire l'acheminement "continu" d'informations d'un objet producteur vers un objet consommateur (*flux*) ;
- *interface de signal*, recevant les interactions élémentaires ("action atomique") et unilatérales que constituent les *signaux*.

Les opérations sont des interactions "client-serveur" qui peuvent prendre deux formes : les *annonces* (invocation sans retour de résultat) et les *interrogations* (invocation suivie d'une terminaison, i.e. d'une réponse de la part du serveur). Une interaction nécessite une *liaison* entre les objets concernés, éventuellement réalisée par un *objet de liaison* offrant une *interface de contrôle* de la liaison (utile dans le cas d'une interaction complexe — cf. qualité se service, liaison de plus de deux interfaces).

Les notions de *type* et de *signature* sont également très importantes dans RM-ODP. Les interfaces sont en effet fortement typées et il est possible de définir des sous-types d'interface par héritage.

Les objets de traitement ont à leur disposition un certain nombre d'*actions*, qu'ils peuvent composer séquentiellement ou en parallèle, avec ou sans synchronisation : création ou destruction d'un objet ou d'une interface (un objet peut avoir plusieurs interfaces), exportation ou importation d'une interface par courtier (cf. courtage, 3.3.6), liaison avec une interface, lecture ou écriture de l'état d'un objet, et enfin toutes les invocations concernant les différentes interfaces (opération, flux ou signal).

3.3.5 Le langage d'ingénierie

Le langage d'ingénierie est destiné à décrire la conception de l'infrastructure d'un système ODP. Il ne concerne pas la sémantique de l'application mais simplement ses besoins en répartition et en transparence de cette répartition.

Les structures d'ingénierie

Les abstractions du langage d'ingénierie déterminent la structure d'un système ODP, en suivant la décomposition suivante (voir figure 3) :

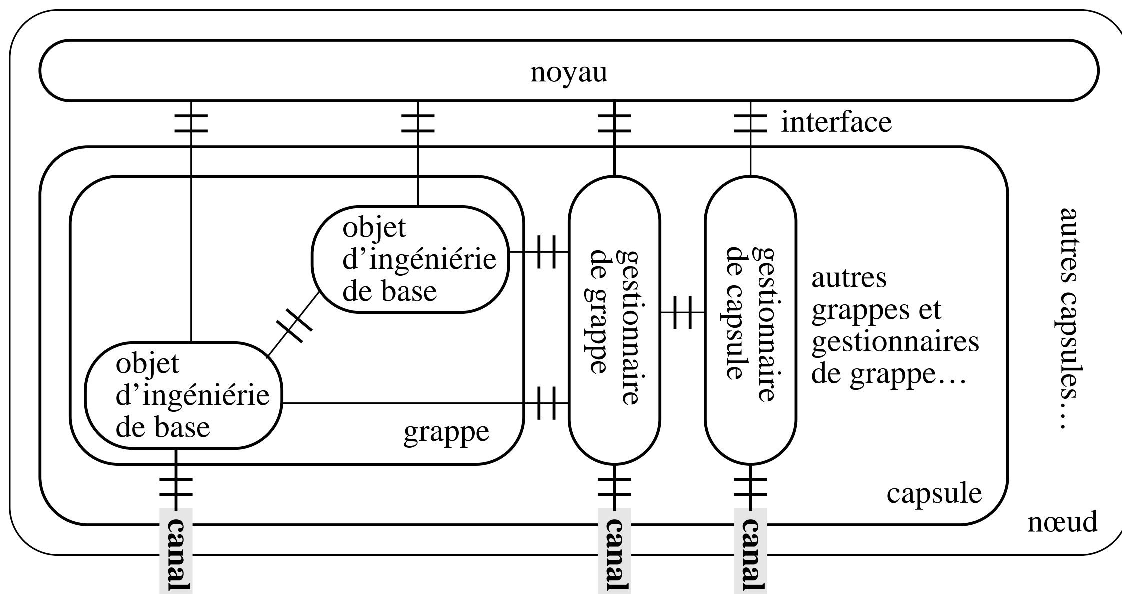


figure 3 : les entités structurant un nœud ODP

- un *noeud* regroupe physiquement un ensemble de fonctions de traitement, de stockage et de communication (typiquement un ordinateur et son logiciel) ;
- chaque noeud possède un et un seul *noyau*, objet assurant la coordination des fonctions du noeud utilisées par les autres objets du noeud (cf. système d'exploitation) ;
- chaque noyau peut supporter plusieurs *capsules*, entités d'encapsulation de traitement et de stockage (e.g. duplication, résistance aux pannes) ;
- une capsule contient des *grappes*, ensembles d'"objets d'ingénierie de base" qui représentent des unités d'activation et de localisation ;
- les *objets d'ingénierie de base* correspondent aux objets de la spécification de traitement, et reposent sur l'infrastructure répartie proprement dite. Celle-ci est assurée par un ensemble d'objets qui participent au fonctionnement des *canaux de communication*.

Les canaux de communication

Les *canaux* correspondent aux liaisons et aux objets de liaison de la spécification de traitement. Leur rôle est de fournir un mécanisme de communication qui contienne et contrôle les fonctions de transparence requises par les objets d'ingénierie de base. Les canaux ne

s'appliquent qu'aux *liaisons réparties*, par opposition aux liaisons dites *locales* qui ne nécessitent aucune transparence.

La figure 4 détaille la composition d'un canal, dans le cas particulier d'une liaison simple client/serveur (un canal peut lier davantage d'objets, le plus bas niveau d'interconnexion étant réalisé par l'objet *intercepteur*). Dans certaines situations, des objets peuvent s'avérer inutiles et être omis. C'est le cas des *objets de protocoles* et *intercepteurs* lorsqu'un canal lie des objets dépendant d'un même noyau. Il en va également ainsi des *objets talons* et *lieurs* quand aucune transparence à la répartition n'est requise.

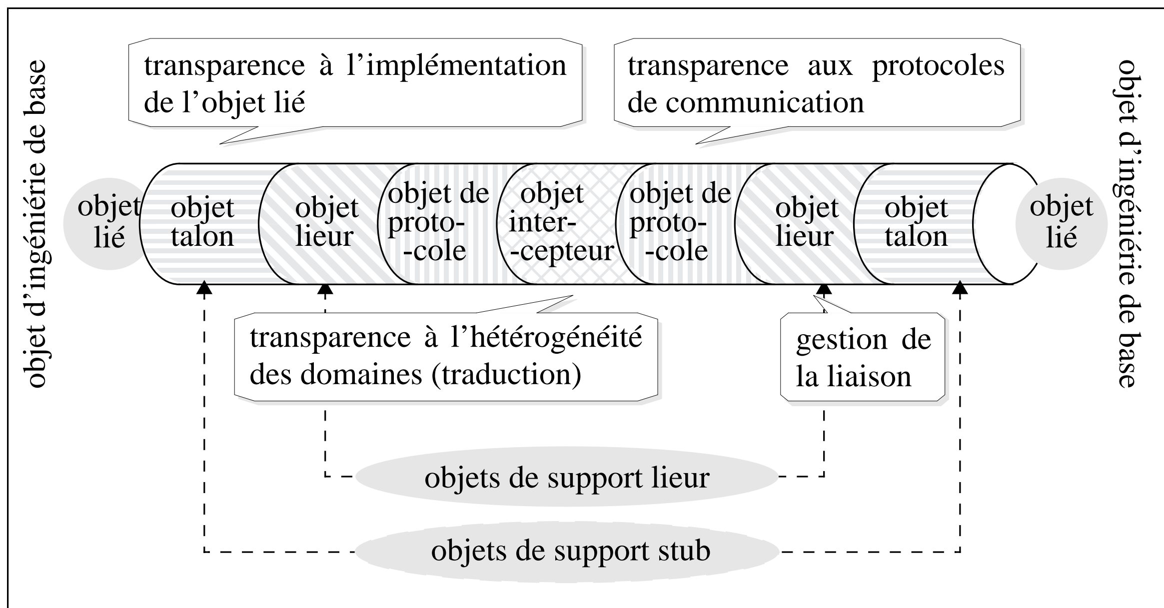


figure 4 : structure d'un canal de communication dans ODP (liaison entre deux objets)

Afin de remplir leur fonction, les objets qui composent un canal peuvent être assistés par des objets d'ingénierie externes au canal. Par exemple, un objet lieur peut interagir avec un objet d'ingénierie “relocalisateur” pour obtenir des informations de localisation. De même, un objet externe peut fournir des fonctions de sécurités à un objet talon.

3.3.6 Les fonctions ODP

RM-ODP recense également un ensemble de fonctions susceptibles de répondre à des besoins issus des points de vue traitement et ingénierie. Ces fonctions sont classées en quatre catégories :

Fonctions de gestion

Les fonctions de gestion des structures d'ingénierie (noeud, capsule, grappe, objet, canaux) sont assurées par le noyau, des objets dédiés (gestionnaire de grappe ou de capsule) ou les objets d'ingénierie de base eux-mêmes. Suivant la structure considérée, elles permettent création, activation, désactivation, migration et destruction.

Fonctions de coordination

Les objets, grappes et capsules peuvent coordonner leurs actions grâce à un certain nombre de fonctions telles que la notification d'événement, le contrôle d'activation, la gestion de groupe, la duplication, la migration, la gestion de transactions.

Fonctions de dépôt et courtage

Les fonctions de dépôt incluent le stockage des types, de la localisation des interfaces, et des déclarations de services utiles au service de *courtage*. Le *courtier* ("trader") permet à un objet client de trouver un objet serveur inconnu à partir d'une description de *service*. L'*export* permet au serveur de déclarer un service particulier ("publicité"), caractérisé par un type d'interface et un ensemble d'*attributs*. L'*import* permet au client d'obtenir une référence sur un serveur fournissant le service requis. La normalisation du courtier ne figure pas dans RM-ODP, mais elle est en cours dans ODP.

Fonctions de sécurité

Les fonctions de sécurité concernent le contrôle d'accès (décision et exécution), l'audit (supervision, collecte d'informations et analyse), l'authentification, l'intégrité des données et la confidentialité.

3.3.7 Les "transparences" d'ODP

RM-ODP indique comment les concepts et les services définis permettent d'assurer les différents points de transparence à la répartition et à l'hétérogénéité :

- La *transparence à l'accès c*onsiste à masquer les différences de représentation des données et de mécanisme d'invocation. Elle est rendue par des conversions ad hoc au sein des objets talons.
- La *transparence aux défaillances* permet d'ignorer une éventuelle défaillance, suivie d'une reprise d'autres objets ou du même objet. Elle est rendue soit par une infrastructure

rendant impossible la défaillance considérée, soit par des fonctions de gestion de point de reprise ou de duplication.

- La *transparence à la localisation* exprime la possibilité pour un objet d'accéder à des interfaces sans utiliser d'information sur leur localisation dans l'espace.
- La *transparence à la migration* se traduit par l'aptitude du système à changer la localisation d'un objet sans introduire de perturbation. Cette transparence utilise la fonction de migration, qui doit prendre en compte un *schéma de mobilité* spécifiant des contraintes particulières (sur les interactions, l'exécution et la sécurité). Puisque l'unité de localisation est constituée par la grappe, cette transparence à la migration pour un objet implique une transparence à la relocalisation pour la grappe à laquelle il appartient.
- La *transparence à la persistance* représente la possibilité de désactiver un objet et de réactiver d'autres objets ou le même objet sans perturber le système. Elle utilise la fonction d'activation et de désactivation au niveau d'une grappe, conformément aux contraintes d'un *schéma de persistance* qui spécifie l'utilisation de fonctions de traitement, de stockage et de communication. En outre, ceci implique une transparence à la relocalisation des canaux liés à la grappe.
- La *transparence à la relocalisation* permet de relocaliser une interface sans perturber les interfaces liées avec elle. Ceci implique une propagation d'information sur les changements de localisation d'objets vers les relocateurs, ainsi que des échanges supplémentaires entre objets lieurs au sein d'un canal.
- La *transparence à la duplication* masque l'utilisation d'un groupe d'objets de comportements compatibles pour prendre en charge une interface. Elle est obtenue par application de la fonction de duplication et *d'un schéma de duplication* fixant des contraintes en termes de disponibilité et de performances de l'objet.
- La *transparence aux transactions* permet de cacher la coordination nécessaire des activités au sein d'une configuration d'objets pour en assurer la cohérence. Les transactions sont définies par un *schéma dynamique* et *un schéma d'invariant* qui constituent le *schéma transactionnel*. Celui-ci est appliqué par la fonction de transaction. Des opérations de reprise et de point de reprise concernant l'état de l'objet sont nécessaires.

3.4 L'architecture CORBA de l'OMG

3.4.1 Présentation générale

Comme l'explique [ORF 96], l'Object Management Group a été créé en 1989 afin de spécifier une infrastructure de communication orientée objet véritablement ouverte et non propriétaire. Les objectifs premiers sont de promouvoir une approche objet susceptible d'améliorer la réutilisabilité, la portabilité et l'interopérabilité dans un environnement réparti et hétérogène [CORBA2]. Regroupant à l'origine 11 organisations, l'OMG compte désormais plus de 500 membres, parmi lesquels figurent les ténors de l'industrie informatique (Sun, IBM, Digital, HP, Novell, Microsoft…). L'historique des travaux de l'OMG est détaillé par la figure 5, reprise de [VAN 94].

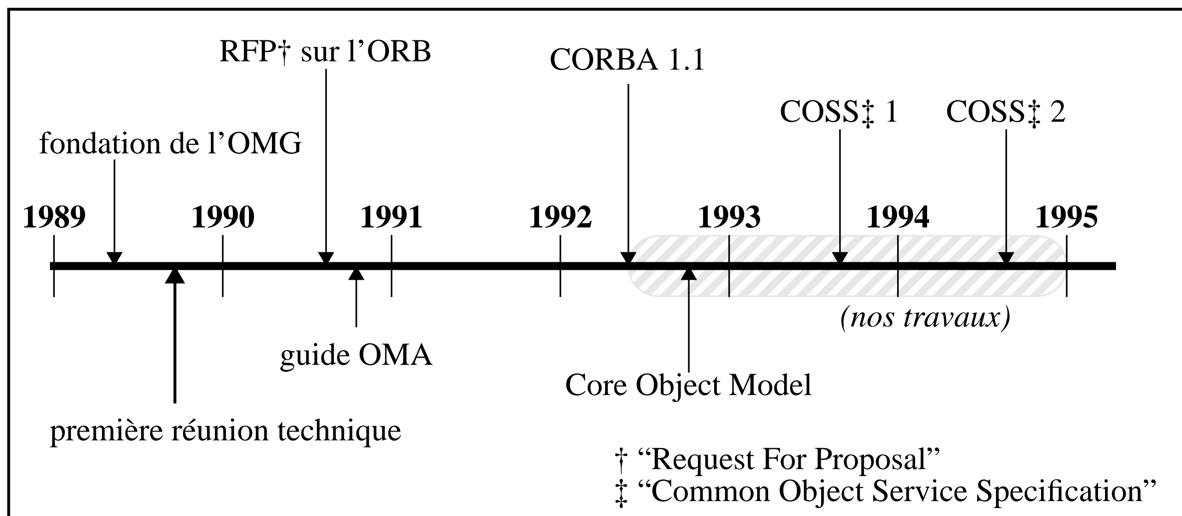


figure 5 : historique des travaux de l'OMG

Les standards de l'OMG définissent :

- un modèle d'objet ;
- un modèle de référence d'architecture pour la gestion des objets, reposant sur la définition d'un Object Request Broker ;
- une architecture commune d'ORB, baptisée CORBA (Common Object Request Broker Architecture) ;
- des services destinés à la gestion "système" des objets ou aux applications.

3.4.2 Le guide OMA

<u>Le modèle objet</u>

Dans le guide OMA (Object Management Architecture), l'OMG définit un modèle objet de base (Core Object Model) qui peut être complété par des extensions spécifiques à des domaines technologiques particuliers, sous réserve de compatibilité avec ce modèle. En associant le modèle de base et certaines extensions compatibles (on parle alors de composants), on crée un profil correspondant à un domaine d'application particulier.

La version 92 du modèle de base décrit précisément les concepts suivants : objet, opération, signature, nom d'opération, paramètre, type de paramètre, résultat d'opération, non-objet, interface d'objet, type, héritage d'interface, requête. Un type d'objet est défini par l'ensemble des opérations qu'il peut traiter. Une classe définit une implémentation possible d'un type d'objet, par des méthodes et une représentation de l'état de l'objet. Les opérations sont regroupées en interfaces héritables.

La version 95 définit une extension — nommée Core 95 — du modèle Core 92, en approfondissant ou introduisant certains concepts afin d'améliorer la portabilité et l'interopérabilité : identité d'un objet, interface multiple sur un objet, extension de la définition des interfaces (aspect dynamique), interface de flot, liaison, services.

<u>Le modèle de référence d'architecture</u>

L'architecture OMA est basée sur le concept d'ORB. Comme le montre la figure 6, 4 composants sont définis :

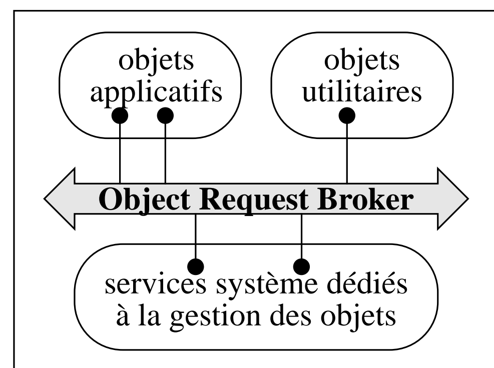


figure 6 : le modèle OMA

* l'*Object Request Broker*, qui fournit tous les mécanismes d'échanges transparents (cf. répartition et hétérogénéité) de requêtes et de résultats ;

* les *services objets*, qui concernent la gestion des objets du système (nommage, cycle de vie, transactions, persistance...) ;

* les *fonctions communes* ("common facilities"), qui offrent des services génériques susceptibles d'être partagés par plusieurs applications (serveur d'impression, aide en ligne, messagerie électronique...) ;

* les objets applicatifs.

3.4.3 Spécification de CORBA

L'architecture CORBA est issue de travaux menés conjointement par l'OMG et l'X/Open[1]. Le manuel [CORBA2], deuxième mouture du standard ("CORBA 2"), se consacre à la spécification du modèle de base ("core"), puis de l'interopérabilité, et enfin de l'implémentation des objets avec des langages particuliers.

Le modèle objet de base ("core")

Le modèle de base précise tout d'abord le modèle d'objet CORBA, qui est issu du modèle abstrait du guide OMA.

Un *objet* est une entité encapsulée et identifiable, capable de fournir un ou plusieurs *services* pour répondre à des *requêtes* émanant de *clients*. Une requête spécifie une *opération*, un objet cible, d'éventuels *paramètres* et un *contexte* optionnel. Il peut y avoir des paramètres d'entrée ("*in*"), de sortie ("*out*") ou mixtes ("*inout*"), ainsi qu'une valeur de retour ("*result value*"). Lorsque des conditions considérées comme anormales se produisent, une requête peut être interrompue et retourner une *exception*.

Une opération est une entité identifiable qui représente un service pouvant faire l'objet d'une requête. Une opération est caractérisée par une *signature*, qui spécifie, s'il y a lieu :

- l'ordre, le type et le mode (i.e. in/out/inout) des paramètres ;
- le type du résultat (qui est un paramètre out particulier) ;
- les exceptions susceptibles de l'interrompre ;
- une information dite de contexte, susceptible de modifier la réalisation du service.

La signature contient également une sémantique d'exécution qui permet de préciser si l'opération doit être invoquée en mode "*at-most-one*" ou "*best-effort*". Dans le premier cas, l'opération est soit exécutée une fois en cas de succès, soit abandonnée si une exception se produit. Dans le second cas, le client ne se synchronise jamais avec l'exécution de l'opération, et ne peut donc ni recevoir de résultat, ni même savoir si l'opération s'est déroulée normalement (i.e. sans exception).

1. X/Open est une organisation internationale indépendante ayant pour mission de favoriser la réalisation pratique de systèmes ouverts, à partir de standards existants ou émergents, par le biais de spécifications définissant le CAE X/Open (Common Applications Environment).

Une *interface* décrit un ensemble d'opérations qu'un client peut invoquer sur un objet. Ceci permet de définir le concept de *type* d'interface, que l'on spécifie à l'aide du langage IDL (*Interface Definition Language*). Une interface peut également posséder des *attributs*, ce qui équivaut à la définition de deux opérations, l'une de lecture et l'autre d'écriture.

l'IDL

Le Langage de Définition d'Interface permet de déclarer des interfaces supportant l'héritage multiple, des exceptions, ainsi que des types construits : enregistrements ("*struct*"), enregistrements à champs variables ("*union*"), séquences (succession d'un nombre fixe, borné ou quelconque de valeurs de même type) et énumérations ("*enum*"). Pour chaque interface, les opérations, les paramètres de ces opérations avec leur type, mode et ordre, ainsi que les attributs, sont déclarés suivant une syntaxe dérivée de C++. Des mots clés permettent en particulier de spécifier le mode (*in*, *out* ou *inout*) des paramètres, les exceptions possibles, le style d'exécution ("*oneway*" pour une exécution "*best-effort*"), ou la protection en écriture d'un attribut ("*readonly*").

L'IDL permet ainsi de décrire des services indépendamment du langage utilisé pour leur implémentation. Des compilateurs spécifiques au langage d'implémentation du client ou du serveur utilisent cette description pour générer les composants réalisant le lien avec l'ORB. On parle de "*mapping*" IDL-C, IDL-C++, IDL-Smalltalk, etc. Ces composants sont détaillés dans l'architecture CORBA proprement dite.

Structure de l'ORB

La figure 7 détaille la structure définie par CORBA pour prendre en charge les requêtes émises par un client à destination d'une implémentation d'objet. Un client émet une requête en invoquant :

- soit l'*interface d'invocation dynamique*, qui lui permet de construire dynamiquement une requête correspondant à n'importe quelle opération ;
- soit les *souches clients* ("client stubs"), pré-construites pour une interface donnée (cf. déclaration IDL du serveur) et un langage client particulier.

Côté serveur, les *squelettes* se chargent de transformer les requêtes en appels spécifiques sur une *implémentation* d'objet, et de renvoyer les résultats. De façon analogue au côté client, on trouve des *squelettes statiques*, construits à partir d'une description IDL, et des *squelettes*

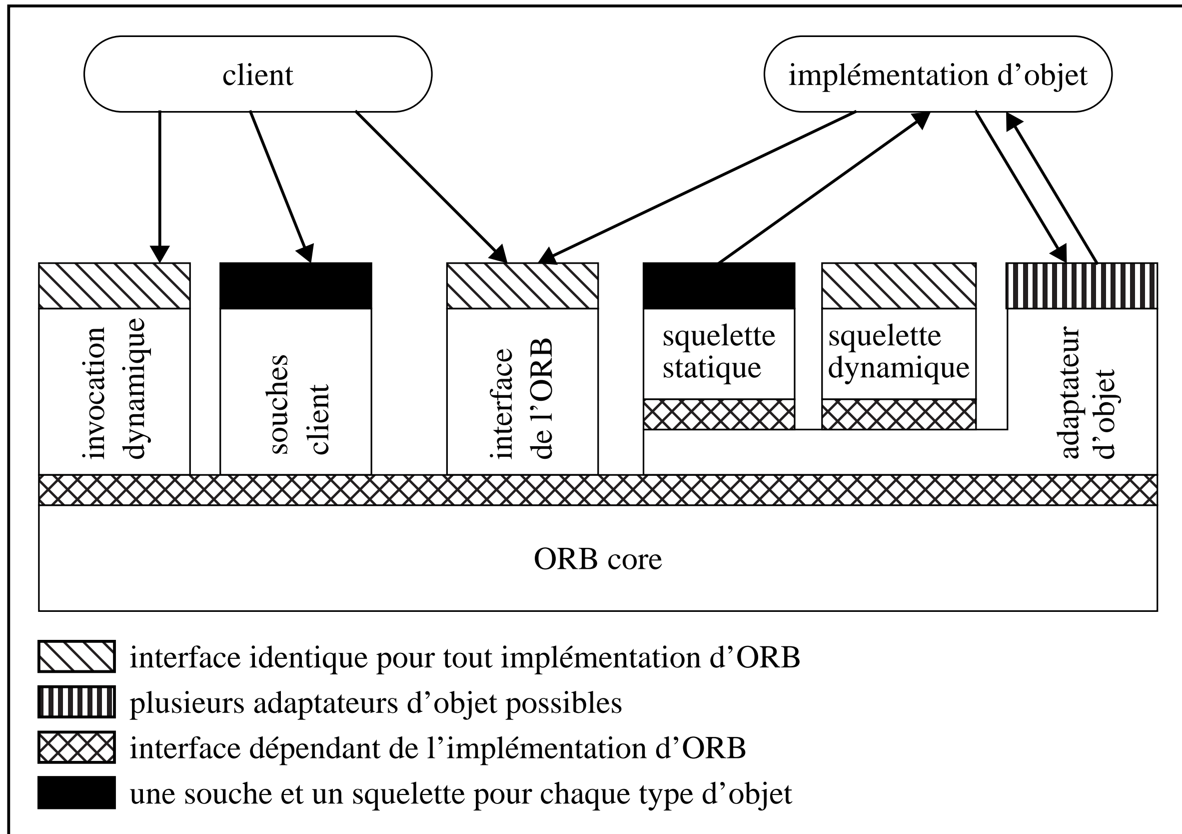


figure 7 : structure des interfaces d'un ORB

dynamiques, capables de prendre en compte n'importe quelle requête pour un langage d'implémentation donné.

Les squelettes reposent sur les services d'un *adaptateur d'objet* qui assure :

- la gestion des références d'objet,
- l'enregistrement des implémentations,
- l'activation de ces implémentations suivant différentes politiques (e.g. création d'un serveur par requête, par client ou partagé),
- l'invocation d'opération par l'intermédiaire des squelettes.

L'interopérabilité

La définition du modèle objet, du standard d'architecture CORBA et de certaines interfaces est insuffisante pour garantir la compatibilité entre différents produits conformes. Par conséquent, CORBA 2 accorde une large part à la spécification d'une interopérabilité entre produits conformes, ainsi qu'avec DCE. L'interopérabilité repose sur l'adoption d'un protocole commun, voire sur l'utilisation de passerelles.

Le protocole général d'interopérabilité **GIOP** ("General Inter-Orb Protocol") possède une version spécifique à TCP/IP, baptisée **IIOP** ("Internet Inter-Orb Protocol"). L'interopérabilité repose également sur la spécification d'un format de codage de l'**IOR** (Interoperable Object Reference), référence unique d'objet.

Services objets

Parallèlement à la question de l'interopérabilité, celle de la portabilité entre produits conformes se pose également. Les problèmes rencontrés dans ce domaine tiennent essentiellement à l'insuffisance des services objets CORBA, qui pousse les industriels à développer des solutions propriétaires [ROY 94]. Face à cette problématique, plusieurs groupes de **COSS** ("Common Object Service Specification") apportent des services de niveau système pour les objets : cycle de vie, nommage, persistance, notification d'événement, transactions, contrôle d'accès concurrents, sécurité, temps global… Définis par leur interface IDL, ils s'intégrent directement dans l'architecture CORBA.

Fonctions communes et objets de métier

Afin d'assurer une factorisation de fonctionnalités génériques de niveau plus élevé que les services objets, la définition de *fonctions communes* ("common facilities") et d'*objets de métier* ("business objects") est en cours. Ces différentes appellations traduisent une augmentation de sémantique, depuis le système vers l'entreprise. Les services rendus seront également représentés par des interfaces IDL et accessibles par requête ORB.

Les fonctions communes visent à permettre le partage de fonctionnalités transversales aux applications (e.g. impression, aide en ligne…) ou plus spécifiques à un domaine. La première RFP[2] en la matière concerne les données et la présentation des documents composites. Dans le futur, les fonctions communes s'intéresseront à l'interface utilisateur, à la gestion de l'information, à l'administration des systèmes, au génie logiciel…

Les objets de métier constituent les composants les plus proches des entreprises. Ils ont pour objectif de permettre la définition d'applications par un simple assemblage.

---

2. "Request For Proposal" : document initialisant un procédure de spécification commune, par la description d'un certain nombre de besoins bien définis, auquel les membres de l'OMG peuvent répondre par des propositions de solution.

3.5 Le système COOL

3.5.1 Le micro-noyau Chorus

La technologie “micro-noyau”

L’approche micro-noyau vise à construire des systèmes d’exploitation portables et modulaires suivant une architecture séparant le matériel, les services génériques et la “personnalité”. Le principe fondamental consiste à encapsuler la gestion bas niveau du matériel par un noyau de taille réduite. Celui-ci est utilisé par un certain nombre de serveurs, destinés à fournir les fonctions génériques typiquement offertes par les systèmes d’exploitation (e.g. système de fichier, mémoire virtuelle, ordonnancement de tâches). Ainsi, ces services peuvent être implémentés de façon portable et modulaire. Cette modularité permet un découpage de la complexité en entités indépendantes et une configuration dynamique du système global. Enfin, une (voire plusieurs simultanément) personnalité(s) particulière(s) de système d’exploitation assure(nt) l’interface de ces serveurs vis-à-vis des applications.

En intégrant des fonctionnalités de communication directement au cœur du micro-noyau, cette technologie est particulièrement adaptée à la répartition et à l’intégration de systèmes, qui peuvent mêler des composants aussi divers que des PABX[3] et des stations de travail UNIX dans une architecture unifiée.

Selon [ARO 95], les atouts de la technologie micro-noyau détermineront sans aucun doute la conception des systèmes d’exploitation de demain. Certes, les bénéfices en termes de portabilité, de modularité et de communication ont aussi un coût en matière d’efficacité. Mais la complexité croissante des systèmes d’exploitation[4], le besoin de répartition et d’intégration, ainsi que l’approche orientée objets, imposeront probablement ce type d’architecture, même si elle se confronte à court et moyen terme à des problèmes de compatibilité avec l’existant et à des réalités commerciales.

Andrew Tanenbaum figure parmi les auteurs de livres qui font le plus autorité sur le plan international en matière de système d’exploitation et de réseaux de communication. En introduction à son exposé d’ouverture de la conférence ICDCS ‘94[5], il a fortement appuyé

3. “Private Automatic Branch eXchange”, autocommutateur numérique de télécommunications.
4. En 1990, Steinar Høistad [HØI 90], directeur général Europe d’Unix International, précisait qu’avec 1,2 million de lignes de code, Unix System V.4 se situait au deuxième rang des projets informatiques du point de vue de l’importance des développements, juste après celui de la conquête de la Lune.
5. 14th International Conference on Distributed Computing System.

l'idée selon laquelle les futurs systèmes d'exploitation répartis devront être basés sur des micro-noyaux. Il a comparé ces derniers à des systèmes d'exploitation "RISC" prenant en charge à très bas niveau la gestion des processus, de la communication inter-processus, de la mémoire et des entrées-sorties. Le très haut niveau de modularité requis pour ces systèmes pourraient même nécessiter des "nano-noyaux".

Les caractéristiques de Chorus

Le micro-noyau Chorus, développé par la société Chorus Systèmes, est issu de travaux de l'équipe Chorus de l'INRIA. Sa structure comprend une couche de gestion du matériel, typiquement non portable, ainsi que trois modules, portables dans une très large mesure, gérant respectivement les communications (IPC Chorus), la mémoire virtuelle et l'ordonnancement des tâches (cf. figure 8).

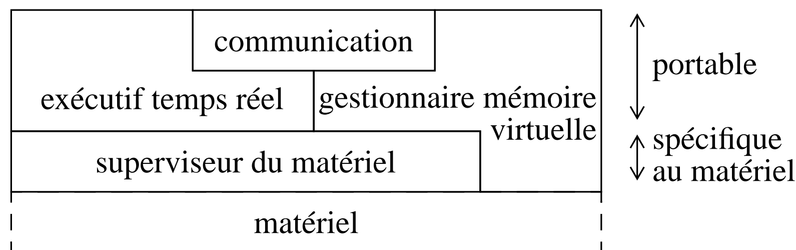


figure 8 : structure du noyau Chorus

En dehors de sa taille réduite (environ 100K pour le noyau Chorus V3), le noyau présente des caractéristiques spécifiques. D'une part, il est résolument tourné vers les systèmes répartis, grâce notamment à des fonctionnalités de communication avancées et efficaces, intégrées au cœur même du micro-noyau : transparence à la localisation, communication synchrone (de type RPC) ou asynchrone avec notion de groupes. D'autre part, il repose sur un modèle d'acteur "multi-thread" spécifique. Les abstractions fondamentales, telles que décrites dans [MET 89], sont les *sites*, les *acteurs*, les *activités*, les *portes* et les *groupes* de portes.

- La notion de *site* fixe la granularité de répartition physique du système Chorus, chaque site devant posséder un micro-noyau. Il s'agit typiquement d'un ou plusieurs processeurs associés de façon étroite avec de la mémoire et d'éventuels périphériques.
- Un *acteur* définit un espace d'adressage et une unité d'allocation de ressources pour l'exécution d'activités. L'acteur constitue une unité de répartition puisqu'il ne peut appartenir qu'à un seul site.

- L'*activité*, unité d'exécution du système Chorus, relève de la notion de “thread”. A la différence des processus (e.g. de type UNIX), l'activité ne possède pas un espace d'adressage privé. Son environnement d'exécution est l'acteur, partagé avec les autres activités qui s'y déroulent en “parallèle” (ordonnancement temps réel, avec priorités, assuré par le noyau). Une activité au sein d'un acteur est donc définie par une pile d'exécution et l'état des registres du processeur.
- Chaque acteur possède une ou plusieurs *portes* de communication servant d'adresse et de boîte aux lettres (stockage en file d'attente) pour recevoir des messages. Ces messages sont envoyés et lus par les activités, de façon transparente à la localisation, via les portes de leur acteur respectif. Une caractéristique intéressante de ces portes est la possibilité de les migrer d'un acteur à l'autre, y compris entre sites différents.
- Enfin, un ensemble quelconque de portes peut être adressé en diffusion (multicast) ou en mode fonctionnel (i.e. seule une porte aléatoire du groupe reçoit le message), par la définition dynamique de *groupes* de communication.

3.5.2 COOL v1, une couche orientée objets sur Chorus

Le SEPT est à l'origine de la spécification de la couche orientée objets COOL v1 (“Chorus Object Oriented Layer”), développée par la société Chorus Systèmes ([LEA 93]) en collaboration avec l'INRIA ([HAB 89]). Une première version de CIDRE, développée en C++ et utilisant des RPC, a notamment permis de mettre en lumière l'utilité d'un support système plus adapté à la répartition et à l'autonomie des entités applicatives. COOL v1 découle donc directement de besoins génériques de répartition logique et physique exprimés dans le domaine de la bureautique communicante.

Un objet de COOL v1 peut être considéré comme une extension du modèle objet, ajoutant des notions de répartition, de communication et d'activité à l'encapsulation des données et du comportement. Au-delà de l'objet lui-même, l'environnement COOL v1 participe également à cette extension, avec un service de nommage réparti et un mécanisme de persistance des objets (cf. figure 9).

En ce qui concerne les communications, elles relèvent de deux procédés différents. D'une part, un objet peut accéder localement à un autre objet par simple invocation d'une méthode de l'interface de celui-ci. Ce type d'appel peut être vu comme un partage d'objet, dont les besoins en exclusion mutuelle sont couverts par la mise à disposition d'objets sémaphores. D'autre part, les objets peuvent s'échanger des messages via une porte de communication personnelle,

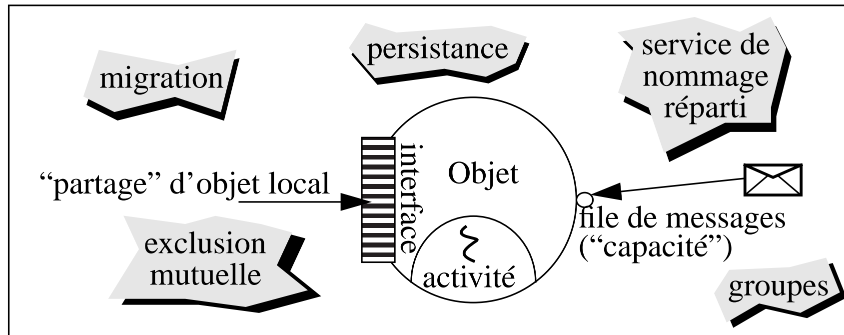


figure 9 : modèle d’objet COOL v1

appelée *capacité*, soit de façon bloquante (i.e. attente d’une réponse), soit de façon totalement asynchrone. Un service de nommage réparti permet aux objets de désigner leur capacité par un nom symbolique choisi.

Par ailleurs, un objet COOL v1 peut posséder une activité propre, être sauvegardé puis restauré via un mécanisme de persistance, et être migré (voire s’auto-migrer) par le biais d’un message à destination d’un autre objet d’un site quelconque. Cependant, la restauration et la migration d’un objet ne permettent pas de conserver l’activité, qui est redémarrée à zéro. Enfin, un objet COOL peut posséder des objets membres et créer dynamiquement d’autres objets COOL. A sa création, un objet COOL v1 sera qualifié d’*actif* s’il possède une activité propre, et de *serveur* s’il possède une capacité.

3.5.3 La mise en œuvre de COOL v1

Implémentation de COOL v1

Dans une large mesure, COOL v1 consiste en une encapsulation orientée objets du système Chorus. Ainsi, les objets COOL sont créés et évoluent au sein d’espaces d’exécution et d’adressage, appelés *contextes*, qui sont en fait des acteurs Chorus. Au niveau de l’acteur, une porte de communication est associée à chaque capacité d’objet, et aux activités des objets correspondent autant d’activités Chorus. Les sémaphores, les communications et les fonctionnalités de groupe reprennent directement les primitives Chorus, de même que la migration d’un objet utilise la migration de porte Chorus.

Le système COOL v1 a été implémenté pour les ordinateurs de type PC avec noyau Chorus natif et sous-système UNIX (e.g Unix SCO pour Chorus/Fusion), mais également pour station de travail Sun, avec un simulateur Chorus fonctionnant au dessus d’Unix. Le système consiste en deux couches : l’une a pour rôle d’adapter et d’étendre les mécanismes de Chorus à un environnement objets, alors que l’autre est dédiée à la prise en compte d’un langage

particulier, C++ en l'occurrence. Chaque site dispose d'un noyau COOL qui rassemble l'essentiel des primitives COOL et assure la gestion mémoire locale des contextes, classes et objets. Au sein de chaque contexte, un *chargeur* d'objets COOL ("run") fournit des environnements d'exécution aux activités internes et les ordonnance.

La structure de contexte et d'objet, détaillée par la figure 10, appelle quelques commentaires et précisions. En effet, COOL v1 présente la particularité de réunir le code et les données initialisées dans un unique segment "text", alors que celles-ci constituent traditionnellement un segment "data" distinct. Par voie de conséquence, les données "statiques" (cf. terminologie du langage C) deviennent ainsi des pseudo-variables de classe partagées par les objets d'un contexte donné. Ce procédé, qui impose une vigilance de la part du programmeur, permet de limiter l'encombrement des objets en évitant de dupliquer certaines données constantes (typiquement des chaînes de caractères, ou plus généralement des tableaux de constantes).

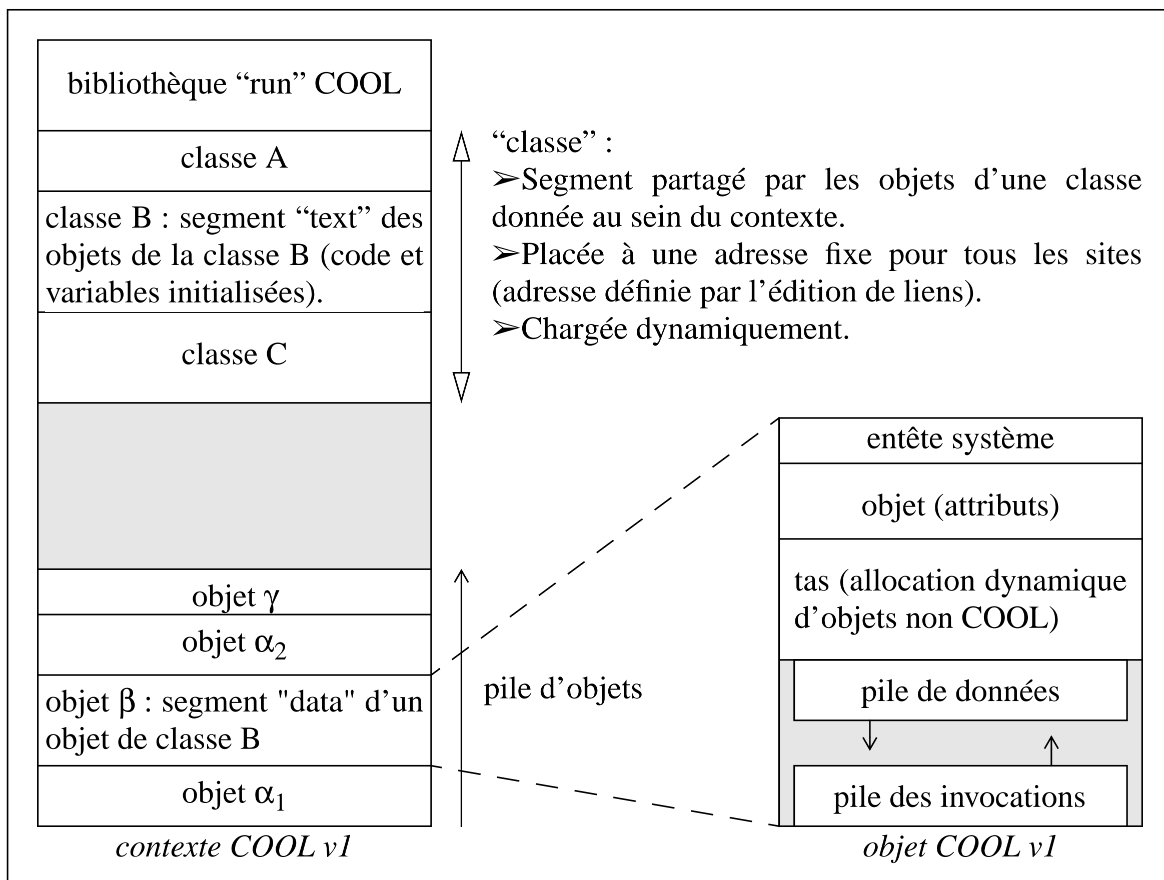


figure 10 : COOL v1 — structure d'un contexte et d'un objet

En ce qui concerne le chargement dynamique du segment "text", il s'effectue :

- soit à partir d'un fichier local, lorsqu'un objet est créé dans le contexte ;

- soit via le réseau depuis le contexte de départ, lorsqu'un objet migre et que le segment de sa classe n'a pas encore été chargé dans le contexte d'arrivée. Ce transfert de code binaire entre sites interdit toute hétérogénéité matérielle pour la migration.

Enfin, chaque nouvel objet qui apparaît dans un contexte donne lieu à une réservation de mémoire au sein de celui-ci. Classiquement, un objet possède une pile d'exécution, une pile de données, un tas pour les allocations dynamiques, et une zone de données pour ses attributs. Une entête système précise la localisation de ces différents espaces, ainsi que les points d'entrée du constructeur, du destructeur et de l'activité de l'objet.

Environnement C++

La partie de COOL v1 dédiée au langage C++ propose une classe "`COOL`", dont une classe C++ doit hériter afin de posséder les caractéristiques exposées en 3.5.2. Cette approche engendre deux catégories d'objets :

- les "objets COOL", vus par l'exécutif ;
- les objets C++ standard, internes aux objets COOL.

Les principales méthodes de la classe `COOL` sont présentées par le tableau 11. Certaines options ou variantes de celles-ci permettent de dupliquer un objet (lors d'une sauvegarde, restauration ou migration), de migrer ou détruire un objet avec ses objets COOL membres, de diffuser un message à tous les membres d'un groupe ou d'en atteindre un membre quelconque (mode "fonctionnel"). En plus de ces méthodes, COOL v1 fournit des outils à travers le langage C++, pour définir des objets COOL membres, des pointeurs relogeables (indispensables dans le cas d'un objet nomade), et des objets sémaphores C++.

tableau 11 : résumé des principales méthodes de la classe COOL

méthode	fonctionnalité
manipulation élémentaire des objets	
`ref_objet->m()`	invoque la méthode `m()` sur l'objet `COOL` de référence `ref_objet` présent dans le même contexte que l'objet appelant.
`objectCreate()`	crée un objet `COOL` dans le contexte de l'objet appelant.[a]
`objectDelete()`	détruit un objet `COOL` dans le contexte de l'objet appelant.[b]
communication	
`objectCall()`	envoie un message vers une capacité et attend une réponse.
`objectReceive()`	lit le message suivant sur sa capacité (boîte aux lettres) ; sert également à accueillir un objet éventuellement attaché au message.
`objectReply()`	envoie un message en retour à un `objectCall()` ; permet également de migrer vers l'objet à l'origine de cet appel.
`objectSend()`	envoie un message sur une capacité ; sert également à migrer vers l'objet lié à cette capacité en s'attachant au message.
service de nommage	
`nameAdd()`	enregistre une capacité sous un nom symbolique.
`nameLookup()`	pour obtenir la capacité associée à un nom symbolique.
`nameRemove()`	retire un nom symbolique du service de nommage[c].
groupes	
`groupAllocate()`	crée un nouveau groupe de communication[d].
`groupInsert()`	ajoute une capacité d'objet dans un groupe[e].
`groupRemove()`	retire une capacité membre d'un groupe.
persistance	
`objectRetrieve()`	réinstalle un objet `COOL` sauvegardé.
`objectSave()`	sauvegarde un objet `COOL`.

a. Ne se substitue pas à l'opérateur `new` destiné à la création d'objets C++ standard.
b. Ne se substitue pas à l'opérateur `delete` destiné à la destruction d'objets C++ standard.
c. Le service de nommage réparti est implémenté de telle sorte que le retrait d'un nom ne peut se faire que sur le site où il a été précédemment enregistré.
d. La création d'un groupe engendre une capacité de communication qui représente ce groupe. Celle-ci peut être enregistrée dans le service de nommage.
e. Il n'est pas possible d'insérer une capacité de groupe dans un groupe.

Bien entendu, la notion de constructeur et de destructeur C++ est conservée de façon standard dans COOL v1. De plus, la présence d'une méthode `main()` dans une classe permet

de définir des objets actifs. Cette méthode est automatiquement exécutée à chaque installation d'un objet, que se soit après création, après migration ou après restauration (cf. persistance), ce qui traduit l'impossibilité de migrer une activité. Toutefois, la distinction entre une création et une réinstallation est possible grâce au constructeur. Le programmeur peut donc gérer l'initialisation et la reprise de l'activité d'un objet par le biais d'attributs d'"état" alliés à une politique de migration ad hoc.

3.5.4 COOL v2 : la prise en compte des standards

Plus qu'une nouvelle version, COOL v2 apporte une révision complète du modèle d'exécution et de communication des objets COOL. Ainsi, de nouvelles abstractions issues de l'OMG ou d'ODP sont introduites, et l'architecture de communication repose sur CORBA.

le modèle d'exécution

Les objets COOL v2 sont tous passifs. Ils ne sont activés que lorsqu'une de leurs méthodes est invoquée. Les objets sont regroupés en *clusters* (cf. ODP) qui définissent une unité de persistance et de répartition[6]. Les *capsules* (cf. ODP) sont des espaces d'adressage et des unités d'allocation de ressources destinées à accueillir les clusters. Une *activité* représente une suite d'invocations susceptible d'activer plusieurs objets dans plusieurs clusters. Elle se traduit par une *thread* au sein de chaque capsule concernée. Lorsque l'activité affecte un objet distant, on dit qu'elle est *diffusée*. Une activité est une entité manipulable qui propose une interface de contrôle.

l'architecture de communication

COOL v2 homogénéise les modes de communication en substituant l'envoi de message (cf. COOL v1) par une généralisation de l'appel de méthode. Unique mécanisme d'interaction, il se décline sous une forme synchrone (attente de la fin de l'exécution) ou asynchrone[7]. L'architecture repose sur un ORB (COOL-ORB) qui permet d'invoquer un objet indépendamment de la localisation et de l'hétérogénéité. Ce mécanisme repose sur la notion d'*interface*, décrite en IDL (cf. OMG).

Pour invoquer une opération sur un objet "serveur", l'objet appelant ("client") doit utiliser la référence d'une interface associée. En cas d'anomalie (e.g. problème réseau, disparition de

6. Par rapport à COOL v1, cette entité de granularité supérieure à l'objet permet de manipuler conjointement un ensemble d'objets sans gérer une arborescence d'objets membres.
7. Il est alors impossible de récupérer des paramètres de sortie.

l'objet serveur), le système engendre une *exception*, dont les caractéristiques peuvent être récupérées via un paramètre facultatif d'"environnement CORBA".

<u>les services objets de COOL-ORB</u>

COOL-ORB offre un certain nombre de services objets standards, tels que le nommage symbolique, qui permet d'associer un nom à une référence d'interface, ou l'exclusion mutuelle (verrous, sémaphores...). A cela s'ajoute le service de groupes de communication, qui n'est pas encore spécifié dans CORBA.

Le fonctionnement des groupes s'affranchit désormais du support de Chorus afin d'offrir un service de niveau ORB, adapté aux modes d'invocation correspondants, et ce sur de multiples plates-formes. Un groupe est une entité à part entière, manipulable via une interface, qui rassemble les références d'interface de ses membres. Son rôle est d'assurer de façon transparente l'invocation des membres et le retour du résultat. Cette tâche est paramétrée par :

(1) une politique de distribution de l'appel parmi les membres du groupe (diffusion ou mode fonctionnel) ;
(2) une politique de collecte des résultats (retour de la première réponse reçue ou de la première exception).

<u>les spécificités de COOL v2</u>

Malgré la perte de spécificité due à l'adhésion au standard CORBA, COOL v2 conserve des caractéristiques propres. En effet, si COOL-ORB est disponible sur des systèmes variés (SunOS 4, Windows NT, Solaris 2, Chorus Fusion pour Unix SCO), COOLv2 n'est disponible qu'avec l'ORB basé sur le micro-noyau Chorus (système Chorus Fusion). Ainsi, les fonctionnalités de persistance et de migration de cluster, offertes par COOL v2, sont directement issues de Chorus. Elles ouvrent la voie à une robustesse accrue : persistance des objets serveurs et déplacement en cas de défaillance de la machine hôte (résistance aux pannes), placement optimal en fonction des objets clients (efficacité de la communication).

3.6 Conclusion

Les bénéfices à tirer de l'approche des systèmes répartis orientés objets concernent l'ensemble de l'industrie informatique : accroissement de la puissance globale mais réduction de la complexité locale, aptitude à la modélisation voire à la simulation, réutilisabilité,

interopérabilité en environnement hétérogène. Certains points sont tout particulièrement pertinents dans le cadre du groupware et des systèmes d'information de l'entreprise :

- il s'agit de problèmes répartis logiquement et physiquement par nature ;
- la généralisation de postes de travail de type “PC”, aux capacités croissantes, favorisent les implémentations de type client-serveur ;
- les besoins d'interopérabilité dans des environnements hétérogènes sont déterminants.

Enfin, l'approche objets répartis s'avère nécessaire pour permettre l'adaptation rapide des systèmes d'information aux réorganisations et à l'évolution continuelle de l'entreprise et des affaires. De plus, comme l'estime [STI 94], le non-déterminisme des applications futures rendront indispensable l'intégration de fonctionnalités intelligentes (apprentissage, reconnaissance de formes, systèmes à base de règles, résolution de contraintes). Cette nécessité d'objet et d'intelligence en environnement réparti nous conduit naturellement à nous intéresser à l'intelligence artificielle distribuée et à la notion d'agent…

Chapitre 4

L'approche "multi-agents"

"All real systems are distributed"
F. Hayes-Roth, repris dans [GAS 91]

4.1 Introduction

Qu'entendons-nous par le terme "agent" ?

Tel est le point essentiel que ce chapitre va tenter de préciser. Ne le cachons pas, l'agent est aujourd'hui à la mode. Mais si de plus en plus de monde en parle, le mélange de différentes approches et de différents niveaux d'abstraction transforme parfois les débats en tours de Babel vacillant sur des questions de vocabulaire. Pourtant, il existe des problématiques communes.

La notion de système multi-agents a été mise en évidence par une discipline assez récente baptisée "Intelligence Artificielle Distribuée". Directement issue de l'Intelligence Artificielle, l'IAD est née de l'apparition de systèmes multiexperts, dont l'objectif était de rendre moins complexe la résolution de certains types de problème grâce à une répartition de la connaissance entre plusieurs entités spécialisées. L'indépendance de ces entités a soulevé des questions de coopération, négociation, synchronisation, et conduit à la notion de Système Multi-Agents.

De nouvelles problématiques concernant la communication, mais aussi la représentation d'autrui, des engagements, des croyances et des connaissances, ont drainé d'autres

communautés scientifiques qui ont apporté leur savoir, leur expérience, leurs préoccupations et leurs champs d’expérimentation. C’est particulièrement le cas de la sociologie et de l’éthologie[1] des espèces sociales, très concernées par les phénomènes de groupe, mais également de la linguistique, déjà liée à l’Intelligence Artificielle par les recherches en traitement automatique du langage naturel. Ces rapprochements ont introduit une première équivoque à propos du terme “agent”, employé pour désigner une entité soit quelconque (physique ou abstraite) soit, de façon plus restrictive, électronique ou informatique.

Aujourd’hui, le terme “agent” est repris par les ingénieurs en informatique et en télécommunications, pour désigner une entité informatique servant d’intermédiaire actif entre des utilisateurs et/ou des composants d’un système d’information. Dans ce domaine se mêlent en proportion variable des notions d’autonomie, de comportement, voire d’intelligence, mais également de nomadisme car l’idée de répartition est centrale, l’agent devant être idéalement placé pour les services qu’il doit rendre.

Enfin, de façon transversale, le souci d’implémentation conduit toutes ces communautés à s’intéresser à des modèles d’implémentation et à l’aspect génie logiciel, l’agent pouvant être considéré comme une évolution du modèle objet.

4.2 L’Intelligence Artificielle Distribuée et le concept d’agent

4.2.1 De l’Intelligence Artificielle à l’IAD

A l’origine, l’Intelligence Artificielle (IA) s’était donnée pour but de résoudre des problèmes complexes par des machines, en cherchant à transposer le raisonnement, le savoir-faire, l’expérience qu’un humain utilise lorsqu’il effectue une tâche ([ERC 93]). En s’inspirant d’observations du “monde réel” (sciences cognitives), ou de modèles théoriques (logiques), l’IA recherche et propose des modèles de représentation des connaissances et des techniques de résolution de problème. A travers des approches symboliques (représentation explicite des connaissances sous forme de faits et de règles), algorithmiques (e.g. vision et traitement d’image) ou sub-symboliques (connexionisme, algorithmes génétiques), l’IA a conduit à la réalisation de machines aux capacités cognitives propres.

1. L’éthologie est la science des comportements des espèces animales dans leur milieu naturel. A travers des études de comportement de colonies de fourmis, termites ou abeilles, l’éthologie des insectes sociaux a permis de mettre à jour des modes de communication et d’auto-organisation permettant une régulation et une adaptation aux perturbations à l’échelle de la colonie.

Mais une approche purement centralisée limite le champ d'application de l'IA, car elle rend difficile la résolution de nombreux problèmes naturellement répartis, hétérogènes et concurrents. La répartition d'une résolution entre des modules spécialisés, s'échangeant des résultats intermédiaires pour parvenir à une solution globale, marque l'apparition du concept d'IAD. C'est ainsi que se sont développés les systèmes multiexperts et les architectures de type “blackboard”, sorte de mémoire partagée permettant la coopération entre des spécialistes (*sources de connaissance*). La répartition est non seulement d'ordre logique, avec une notion de collaboration d'experts, mais également physique (e.g. gestion intelligente d'atelier, planification/exécution/contrôle multirobot).

Dans [GAS 91], Les Gasser dresse un panorama des raisons qui motivent la distribution de l'intelligence artificielle et l'approche multi-agents. Certaines sont à l'origine de l'informatique répartie en général, telles que la répartition géographique naturelle des traitements et des données, ou l'étendue et le niveau de complexité de certains problèmes rendant non viables la conception, la mise en œuvre et la maintenance d'un système monolithique, tant pour des raisons de sémantique que de capacité de traitement. D'autres sont plus directement en rapport avec l'IA, comme le besoin d'intégration de systèmes d'IA existants, ou la recherche de modèles s'inspirant du caractère collectif de l'activité humaine, de l'intelligence et de la résolution de problèmes. De façon synthétique, [HUH 87] énumère six points clés justifiant le développement de l'IAD :

- l'avancée technologique et l'extension des réseaux, permettant une communication à grande échelle ;
- la confrontation à des problèmes dont la répartition est inhérente ;
- le besoin de modularité pour faciliter la conception et l'implémentation ;
- l'effet de synergie, qui permet de résoudre une complexité globale élevée par des entités de conception plus simple ;
- l'apport d'une unité de point de vue, ouvrant la voie à des modèles homogènes d'intégration des utilisateurs et des machines ;
- les théories sociologiques y trouvent un champ d'application et de simulation.

En passant de l'IA à l'IAD, une autre forme d'intelligence est mise en évidence : la capacité à s'organiser pour coopérer. En effet, l'intelligence ne réside pas uniquement dans les capacités cognitives des entités et leur aptitude “volontaire” (i.e. déterministe) à coopérer. Lorsqu'on met en présence un grand nombre d'entités, de leurs interactions naissent parfois

des phénomènes, des comportements qui ne sont ni programmés de façon explicite, ni attendus. Ce qu'on appelle *émergence* constitue une autre forme d'intelligence, une nouvelle voie ouverte par l'approche des systèmes multi-agents.

4.2.2 Les systèmes multi-agents

Les systèmes multi-agents constituent une discipline très récente et très ouverte, sujette à des influences nombreuses et variées, qui n'est pas formalisée de façon consensuelle et exhaustive. Il n'existe donc pas de définitions claires et précises de ce qu'est un agent et de ce qui ne l'est pas, ni de ce qu'est un SMA et de ce qui ne l'est pas. Il n'y a pas non plus de modèle universel de spécification, de conception et de mise en œuvre de SMA. Ceci contribue à rendre floue l'intersection — ou la frontière — avec l'IAD.

Dans ce qui suit, nous ne considérons pas les termes d'agent et de système dans toute leur généralité (informatique, sociologie, éthologie, systémique[2]…), mais dans leur acception d'entité artificielle, informatique. Nous proposons quelques définitions assez générales, partagées dans une large mesure par la communauté IAD/SMA et largement inspirées de [FER 93]. On remarquera que derrière certains mots se cachent des notions qui prêtent à discussion ou constituent des sujets d'investigation à part entière.

Qu'est-ce qu'un agent ?

Un agent est une entité informatique ou électronique plongée dans un environnement sur lequel elle est capable d'agir. Elle possède un comportement autonome conséquence de ses observations, de sa connaissance et des interactions qu'elle entretient avec les autres agents. Commentons les notions importantes de cette définition :

(1) L'aspect *entité autonome* est fondamental. D'ailleurs, comme le propose Jacques Ferber, la notion d'agent pourrait conduire à une technique de génie logiciel succédant à la programmation par objets et qui serait la programmation orientée agents (voir 4.3).

(2) Une *entité informatique ou électronique* : cela ne signifie pas que l'être humain soit exclu du monde des agents. Par exemple, l'approche multi-agents dans le domaine du "Human Computer Cooperative Work" semble tout à fait adéquate. Pour intégrer

2. De nombreux membres de la communauté SMA participent au mouvement systémique, avec d'autres chercheurs d'origines diverses mais également intéressés par la notion de système. Cette discipline vise à établir un modèle cohérent et unique de l'émergence, du fonctionnement et de l'évolution des systèmes ([SCH 92]), qu'ils soient politico-économiques, humains, écologiques, voire symboliques (IAD, vie artificielle).

l'utilisateur dans la modélisation, on peut lui associer un agent qui lui offre une interface plus ou moins évoluée (clavier-écran, technologie multimedia…). On peut alors voir l'utilisateur comme une ressource plutôt "capricieuse" connectée au système par l'intermédiaire de l'agent qui le représente.

(3) L'*interaction avec l'environnement* est un point critique. En effet, on définit l'environnement comme l'ensemble des autres agents et de l'éventuel monde extérieur, mais la question de ce monde extérieur pose problème : est-il toujours nécessaire ? Peut-on le modéliser par un agent ? Par ailleurs, cela pose la question de la perception de cet environnement par un agent : que sont les sens d'un agent, comment les modéliser ? Il en va de même pour les notions d'action et d'effecteur sur l'environnement.

(4) La notion d'agent n'a de sens que s'il y a système multi-agents. Ceci nous conduit naturellement à nous intéresser à la définition des SMA.

Qu'est-ce qu'un SMA ?

Un système multi-agents est une population d'agents en interaction. Il est constitué d'un environnement, d'un ensemble d'objets parmi lesquels on distingue un ensemble d'agents, ainsi que d'un ensemble de signaux émis par les agents et qui se propagent dans l'environnement. Par rapport à la définition de l'agent, deux éléments fondamentaux apparaissent : la question de la *communication*, suggérée par les signaux, et la notion d'*agents en interaction*. Ces deux points font l'objet des parties 4.2.3 et 4.2.4.

4.2.3 La communication

Généralités

Dans le monde réel, la communication passe quasiment toujours par une action sur le monde extérieur : écriture, parole, gestes, traces laissées sur l'environnement. Comme le suggère la définition des systèmes multi-agents, il y a toujours émission de signaux, même involontaire. Par exemple, même lorsqu'on donne une poignée de main, il y a une communication physique directe, mais il y a également une communication secondaire qui se propage dans l'environnement sous forme de signaux lumineux.

Cette remarque nous amène à considérer une autre caractéristique de la communication : son mode d'acheminement. Dans certains cas, l'émetteur de la communication s'adresse à un récepteur précis mais il peut aussi émettre des signaux, laisser des traces sur l'environnement, ce qui lui permet d'atteindre des récepteurs qu'il ne connaît pas. Du point de vue des systèmes

multi-agents, ce type de communication est fondamental, car, si l’on décide d’ajouter un agent dans un système en fonctionnement, il doit pouvoir s’y insérer bien qu’il ne connaisse aucun agent a priori. Bien entendu, pour qu’il puisse communiquer efficacement, il doit partager une même sémantique avec les autres agents.

Jean Erceau insiste beaucoup sur cette notion de *sémantique commune* qui est capitale dans la conception de systèmes multi-agents. Dans le cas des communications en général, les différences de sémantique entre émetteur et récepteur conduisent à des interprétations variées, d’où la notion d’intentionnalité. Une communication est *intentionnelle* lorsque l’émetteur lui confère une signification précise qu’il veut faire connaître au récepteur. Mais, si le cri d’un animal est susceptible d’attirer ses congénères du sexe opposé, il peut en revanche faire fuir les espèces dont il est prédateur. La communication n’est alors pas intentionnelle mais *incidente*. On note qu’une communication intentionnelle donne lieu à une communication incidente lorsque le message (ou une projection du message) est perçu(e) par un récepteur inattendu.

La communication dans les SMA

Le mode de communication le plus répandu dans les systèmes multi-agents est l’envoi asynchrone de messages. Il est généralement facile à mettre en œuvre du point de vue informatique et il peut bénéficier des recherches de la théorie des actes de langage. Toutefois, même en admettant que l’on définisse une sémantique pour tous les agents d’un système et que ces agents communiquent de manière intentionnelle, plusieurs problèmes subsistent :

- qui contacter, comment faire la connaissance d’autres agents ? Par transitivité, un agent peut entrer en relation avec de nombreux agents (on parle de *réseau d’accointances*) mais des problèmes se posent en termes de démarrage, de bouclage et d’évolution. On peut alors recourir à des structures communes dans l’environnement, connues de tous les agents et par l’intermédiaire desquelles les agents peuvent tisser de nouvelles relations.
- comment être sûr que les messages envoyés sont bien arrivés à destination ? La question de la satisfaction de l’acte de langage est un problème théorique intéressant en soit, comme le montre le problème des *généraux byzantins*.

Exposé dans [TAN 92], ce problème concerne tout algorithme réparti sur un environnement de communication non fiable. L’intrigue met en situation deux généraux alliés qui doivent attaquer simultanément un ennemi pour remporter une bataille ; sinon, leur défaite est assurée. Avant de partir à l’assaut, chaque général doit donc être sûr que son homologue va

attaquer en même temps. Pour se synchroniser, ils communiquent par des messagers susceptibles d'être interceptés. Chaque communication étant indispensable (sinon, elle ne serait pas émise), son expéditeur doit être sûr de sa bonne réception par le destinataire, ce qui engendre un échange infini d'accusés de réception tous indispensables (cf. figure 12).

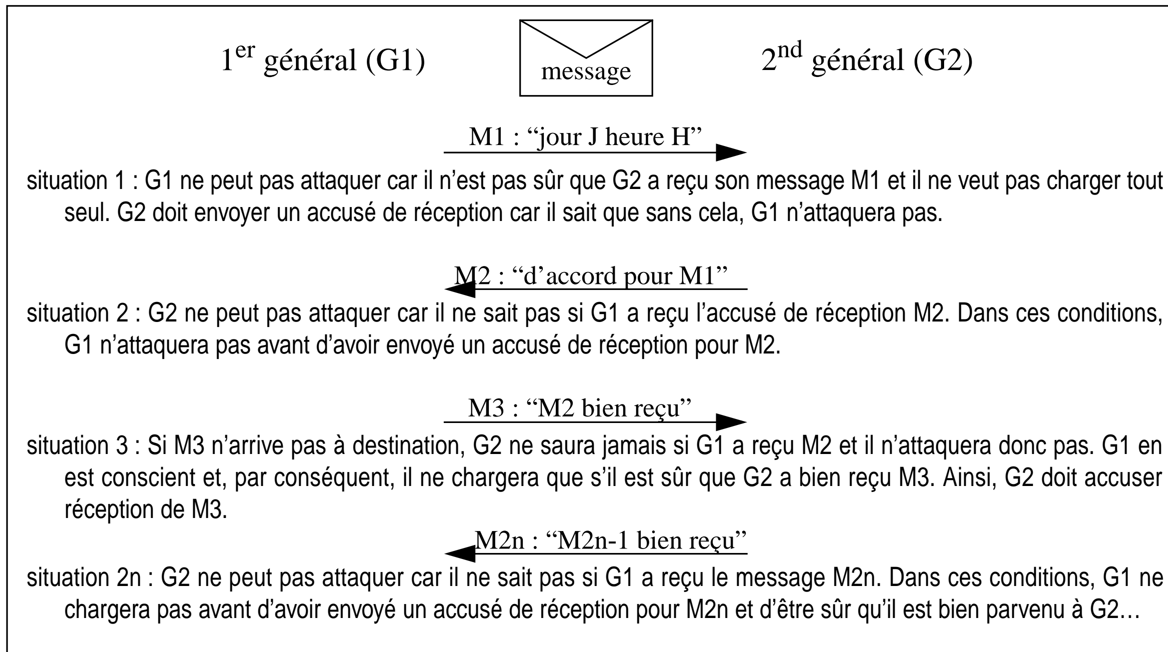


figure 12 : le problème de la synchronisation par message

Même si du point de vue pratique, on considère que les différentes couches (matérielles et logicielles) sur lesquelles on s'appuie sont suffisamment fiables et que l'on peut se restreindre à un certain degré de confirmation, cet exemple montre malgré tout la difficulté de coordonner des actions critiques par messages.

Notion de protocole

Une communication n'est pas un ensemble d'actes de langage isolés mais une suite de messages qui ont des significations explicites et implicites les uns par rapport aux autres. Cette suite est structurée suivant une certaine logique, un certain *protocole* qui peut s'inspirer des études sur les actes de langage (voir [POP 93]) ou de certaines procédures juridiques ou commerciales. Ainsi, le protocole de négociation le plus connu en IAD est celui du réseau de contrat proposé par R.G. Smith dans [SMI 80]. Le principe consiste à diffuser un message d'appel d'offres pour une tâche à accomplir, puis à comparer les offres reçues et enfin attribuer le contrat à l'agent qui a proposé la meilleure offre.

Pour définir ces protocoles, il est avantageux de recourir à des modélisations. L’intérêt d’un formalisme est de pouvoir représenter, puis contrôler ou simuler un mécanisme de négociation. Les automates à états finis permettent de décrire un protocole de façon simple : chaque état correspond à une étape de la conversation. L’inconvénient est que le nombre d’états est combinatoire avec le nombre d’agents impliqués et ce formalisme n’est donc pas adapté lorsqu’on veut représenter un système d’interactions.

Les réseaux de Petri (ou leurs nombreux dérivés) sont souvent utilisés pour modéliser des systèmes concurrents. Par rapport aux automates, on distingue deux intérêts majeurs :

- la possibilité de prendre en compte le parallélisme dans le formalisme ;
- l’aspect modulaire qui permet de construire un réseau à partir de sous-réseaux.

Dans le cas des communications, une telle modélisation d’un protocole au niveau de chaque agent, permet, par interconnexion des réseaux individuels, d’obtenir un réseau global et de simuler une conversation complète. Une des conséquences est la réduction de la complexité du graphe puisque le nombre de places est linéaire en fonction du nombre de participants ; c’est le marquage global qui définit l’état de la communication.

4.2.4 Structure des agents

En s’inspirant du modèle d’agent générique “head+body+communicator” du projet IMAGINE[3], on peut voir un agent comme l’ensemble d’une *tête*, d’un *corps* et d’un canal de communication : la tête pour raisonner, le corps pour interagir avec l’environnement (organes sensitivo-moteurs) et le canal de communication pour échanger des informations avec les autres agents. Ces parties peuvent être plus ou moins développées suivant le type d’implémentation. Or la construction d’un système multi-agents peut s’aborder de deux façons opposées, suivant les motivations du concepteur :

(1) le point de vue de l’ingénieur qui veut créer une application précise. Dans ce cas, le problème posé est le suivant : comment construire les agents pour obtenir tel résultat souhaité ?

(2) le point de vue de l’observateur où la problématique est : que se passe-t-il lorsqu’on met en présence des agents codés de telle façon ?

3. le projet ESPRIT III “IMAGINE” (cf. Chapitre 5) a donné naissance à un langage et à une boîte à outils multi-agents destinés au “Computer Supported Cooperative Work”.

Les agents “cognitifs”

La démarche (1) conduit typiquement à mettre en jeu des agents cognitifs, i.e. des agents évolués, dotés de capacités de raisonnement, dont le comportement est programmé et les interactions prévues. C’est ici que l’on trouve les origines de l’IAD comme évolution de l’IA classique.

Les systèmes à base d’agents cognitifs sont souvent composés d’un nombre restreint d’agents dont le rôle est clairement défini. Pour ces agents, la communication est de haut niveau (symbolique) et la tête, particulièrement développée, peut s’apparenter à un système d’IA classique avec cependant des éléments d’états mentaux supplémentaires (on parle d’*items cognitifs*) :

- croyances (représentation du monde et notamment des autres agents), pour lesquelles le domaine de la logique apporte un support théorique intéressant, en particulier en ce qui concerne les logiques modales ;
- engagements vis-à-vis des autres agents.

Parfois, c’est la démarche (2) qui amène à construire des agents cognitifs, comme dans le cas des études de comportements sociaux basées sur des simulations de micro-sociétés. Ce type d’expérience est très nouveau et tend à se développer suite à des résultats encourageants.

Les agents “réactifs”

Le point de vue (2) conduit plutôt à implémenter des agents très simples afin de pouvoir étudier plus facilement les relations entre la constitution des agents et les phénomènes observés. De tels agents ont une tête très peu développée et entretiennent des communications de bas niveau. Du point de vue de l’IA, les systèmes d’agents réactifs correspondent à une inspiration de type connexioniste.

Le nombre d’agents en interaction doit souvent être conséquent pour que des phénomènes dits *émergents* apparaissent. En effet, lorsqu’on augmente le nombre d’agents dans un système, de nouvelles compétences peuvent apparaître sans qu’elles soient directement apportées par chacun des agents pris individuellement. L’illustration la plus simple est l’*effet de seuil*, ou *effet de masse*, qui fait qu’une nouvelle fonction apparaît dans un système à partir d’un certain nombre d’agents. Mais le plus intéressant à observer est l’émergence d’organisations entre les agents. Par exemple, des simulations informatiques ont montré l’émergence d’une organisation

physique “en chaîne” d’agents programmés individuellement pour le transport d’objets (agents pompiers s’échangeant des sauts d’eau, ou agents “dockers” déplaçant des marchandises).

Les agents réactifs sont l’apanage d’une nouvelle discipline, baptisée *Vie Artificielle*, dont l’objet est de faire interagir des agents très simples dont le comportement est du type “stimulus ⇒ réponse” afin d’observer l’émergence de certains comportements, de certaines organisations, de certaines communications, voire de protocoles. Dans ces systèmes, qui cherchent à “analyser la vie telle qu’elle pourrait être” (Christopher Langton), la communication se fait par l’environnement et les organes sensitivo-moteurs des agents.

L’éthologie des espèces sociales est une source intéressante d’inspiration pour la conception de systèmes à base d’agents réactifs en général et pour la Vie Artificielle en particulier. En effet, des études comme celle du comportement des insectes sociaux ([FRE 93]) permettent d’analyser des systèmes réels composés d’un grand nombre d’entités peu évoluées du point de vue cognitif. Par exemple, les colonies de fourmis sont de très riches sujets d’observation de part la spécialisation variable des membres, les modes de communication et de coopération mis en œuvre, et l’adaptabilité sur le court et le long terme qui en résulte.

Discussion

Cette présentation semble laisser apparaître une classification des agents suivant un critère “réactif/cognitif”. Il ne faut voir dans cette dichotomie qu’un fil conducteur permettant d’exposer de façon structurée les grands principes dont on peut s’inspirer pour concevoir un système multi-agents. En effet, il n’y a pas de rupture brutale entre réactif et cognitif mais une continuité (cf figure 13). De plus, un agent peut présenter une structure mixte avec des aspects cognitifs mais aussi des réflexes purement réactifs à certains stimuli. Ainsi, le modèle cognitif proposé dans [CHA 93] présente le comportement d’un agent en termes de routine, situation familière et non-familière.

Enfin, on peut se poser plus en détail la question de la granularité des agents : par exemple, ne pourrait-on pas considérer qu’un agent cognitif est constitué de plusieurs agents réactifs ? Il existe effectivement des approches qui consistent à concevoir des agents comme des systèmes multi-agents. Du point de vue de l’éthologie, cela revient à dépasser le niveau de la fourmi en tant qu’individu pour appréhender la colonie comme organisme à part entière (notion de *super-organisme*).

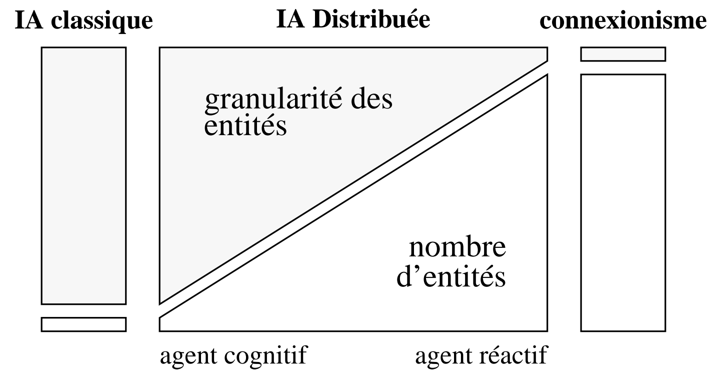


figure 13 : les agents, entre IA classique et connexionisme

4.3 Vers un modèle d'implémentation d'agents

4.3.1 Une programmation "orientée agents" ?

De nombreux chercheurs voient à travers le concept d'agent, un successeur à l'objet. Par rapport à ce dernier, l'agent radicalise les notions d'encapsulation et d'autonomie, avec une indépendance poussée, des capacités cognitives, un comportement — voire une activité — propre. De plus, la question des communications devient incontournable et même centrale. Ceci aboutit au concept de *programmation orientée agents*, présenté dans [SHO 93] comme une spécialisation de la programmation orientée objets. Cette évolution prend en compte des notions de croyances, de décision, d'obligation, et les types fondamentaux de communication sont inspirés de la théorie des actes de langages. Jacques Ferber participe également de façon très active à la recherche d'une méthodologie orientée agents ([FER 95]).

Enfin, [BOU 94] propose un paradigme unificateur de la notion d'agent et d'objet, en se basant sur la double constatation suivante :

- des difficultés dans la construction de systèmes d'information intelligents mettent en lumière des limitations du modèle objet, qui cherche à s'enrichir de nouveaux concepts apportés par les agents (activité autonome, réactivité, capacités cognitives, communication et coopération…) ;
- un fort besoin de méthodes et d'outils induisent un vif intérêt de l'approche agent pour le modèle objet, fréquemment utilisé pour implémenter des agents.

Il en résulte une association agent/objet, le méta-modèle d'un système d'information étant décrit en termes d'agents cognitifs coopérants, auxquels correspondent[4] des objets réactifs de conception (niveau analyse), puis des objets physiques (niveau implantation).

4.3.2 Le modèle acteur

Généralités

Le modèle acteur est souvent considéré comme un modèle d'implémentation particulièrement adapté à la conception de systèmes d'intelligence artificielle ([FER 92], [HAS 92]). Il s'agit d'un modèle de calcul concurrent, basé sur la définition d'entités autonomes qui s'exécutent en parallèle et communiquent de façon asynchrone par messages. L'explicitation de la continuation de ces messages permet de réaliser les structures de contrôle du calcul global. Depuis l'introduction de ce modèle par [HEW 77], de nombreux travaux visent :

- la formalisation ;
- la mise en évidence de propriétés ;
- la définition d'architectures et l'implémentation de langages d'acteurs.

Formalisation

Dans [AGH 88], la formalisation du modèle définit l'acteur comme une entité qui associe un triplet à tout message reçu :

(1) un ensemble fini de *messages émis* vers d'autres acteurs ;
(2) un *nouveau comportement*, qui détermine la réponse au prochain message traité ;
(3) un ensemble fini de *nouveaux acteurs* créés.

Pour chaque acteur du système, les actions correspondant à ce triplet sont exécutées en parallèle et sans synchronisation[5], sans présumer ni des délais ni de l'ordre d'accomplissement. Il en résulte une absence de causalité et un non-déterminisme du calcul global.

Les communications engendrées définissent des *tâches*. Une *tâche* se caractérise par une étiquette unique (identificateur), une adresse désignant l'acteur cible, et un ensemble

4. A chaque transition de niveau, une entité peut engendrer une ou plusieurs entités de niveau inférieur.
5. Le seul mécanisme de synchronisation est basé sur le principe de comportement lié à la continuation d'un message.

d'informations nécessaires au traitement de la tâche. Parmi ces informations, la spécification de l'adresse d'un acteur destinataire du résultat permet de définir la *continuation* du calcul.

Le modèle de communication repose sur un système fiable de messagerie. Chaque acteur possède une adresse explicite, qu'il peut communiquer aux autres acteurs, associée à une boîte aux lettres (fonctionnement de type file d'attente). Un acteur ne peut communiquer qu'avec ses *accointances*, i.e. l'ensemble des acteurs dont il connaît l'adresse. Les accointances d'un acteur incluent au minimum les acteurs qu'il a créés.

Acteurs ou objets ?

Les points (1) et (3) relèvent d'un principe fonctionnel (communication ⇒ action) mais le point (2) relève de la notion d'état, donc d'une sensibilité à l'historique. Ainsi, cette entité indépendante, liant étroitement un état propre à des fonctions invoquées par message, semble s'apparenter au modèle objet. Pourtant, les ressemblances s'arrêtent là.

En effet, l'entité elle-même est de conception toute différente. Ainsi, la notion de comportement pour un acteur ne correspond pas exactement à la notion d'état d'un objet, car un acteur ne possède pas d'attributs et, d'ailleurs, il ne dispose pas d'opération d'affectation. Ensuite, le modèle d'exécution de l'acteur en fait une source autonome d'activités parallèles en nombre variable. Enfin, la notion de continuation permet non seulement de gérer l'asynchronisme des invocations, mais aussi de faire traiter le résultat de l'invocation par un acteur tiers.

4.4 Les agents vus par les informaticiens et les "communicants"

4.4.1 Introduction

De façon récente et croissante, nombreux sont les chercheurs et ingénieurs en informatique ou télécommunications ("communicants") qui empruntent le terme d'agent pour désigner des entités logicielles indépendantes, mises en œuvre de façon pragmatique dans leurs systèmes, dans des couches indifféremment hautes ou basses. Par cette dénomination, ils expriment essentiellement une notion d'intermédiaire actif permettant d'accéder à un certain nombre de services. A travers quelques exemples, nous essayons de mettre en évidence d'autres propriétés caractéristiques liées à l'*agent*.

4.4.2 Les agents répartis

Les agents SNMP ("Simple Network Management Protocol") permettent l'administration à distance d'un réseau de machines ([PUJ 95]). Chaque machine dont on veut pouvoir connaître et/ou modifier la configuration possède un *agent* SNMP, que l'on invoque à partir d'une station de gestion de réseau (*manager)*. Le comportement d'un agent se limite à trois types d'opération :

- envoyer la valeur d'un *objet* à un manager, en réponse à une requête "GET" ;
- modifier la valeur d'un *objet*, conformément à une requête "SET" émanant d'un manager ;
- émettre spontanément des messages d'exception ("TRAP") vers un manager donné.

Les configurations de machines sont manipulées à travers des valeurs d'*objets*. Ces *objets* sont les feuilles d'un arbre décrit sous forme ASN.1 dans une MIB (Management Information Base). La MIB est partiellement normalisée par l'ISO, mais certaines branches peuvent être développées à volonté par les industriels pour les besoins spécifiques liés à leurs produits. A partir de la MIB standard, des MIB privées des industriels, et des protocoles TCP/IP, certains utilitaires sont capables de recenser les machines accessibles à travers un réseau, de rechercher leurs caractéristiques, leur configuration…

A priori, l'agent n'est donc pas intelligent en lui-même, mais l'accès uniforme à des données réparties donne au manager la possibilité d'obtenir une base de raisonnement pour "découvrir" une grande quantité d'informations. Les caractéristiques fortes qui se dégagent de l'agent SNMP sont :

- la répartition, chaque agent étant placé sur la machine à administrer ;
- le rôle d'interface vis-à-vis de ressources, qui offre un accès uniforme à l'information ;
- la notion de service (cf. GET/SET).

4.4.3 Les agents mandataires

Dans le domaine des dispositifs de communication personnels ("Personnal Intelligent Communicator" ou PIC), Bill Atkinson de General Magic prévoit que "les agents pulluleront dans dix ans" ([ATK 93]). Ces agents seront capables de s'interfacer avec les services accessibles à travers le réseau, afin de remplir une mission programmée par l'utilisateur, à l'aide de son PIC.

Dans le domaine de la bureautique communicante et de la gestion électronique des documents, [DUP 94] rejoint cette vision, en présentant l'agent comme "un automate capable d'interpréter une requête et d'analyser les contenus des serveurs présents sur les réseaux pour finalement composer et mettre en forme, selon le script choisi par l'utilisateur, l'information recherchée". Ainsi, "le terme *agent* recouvre et préfigure des mécanismes logiciels ayant des capacités d'autonomie et d'analyse".

4.4.4 Les agents nomades

Dans le cas plus spécifique de General Magic, l'entité programmable qui exécute une mission est dotée de capacités de migration. Cette mobilité traduit le principe du "Remote Programming", qui consiste à créer une entité intermédiaire — l'*agent* — capable de migrer sur le lieu de l'interaction ([WHI 94]). Ce mécanisme s'oppose à l'appel distant de procédure (Remote Procedure Call), où l'interaction entre un client et un serveur distant consiste en un échange de messages de requête et de réponse transitant par le réseau (cf. figure 14).

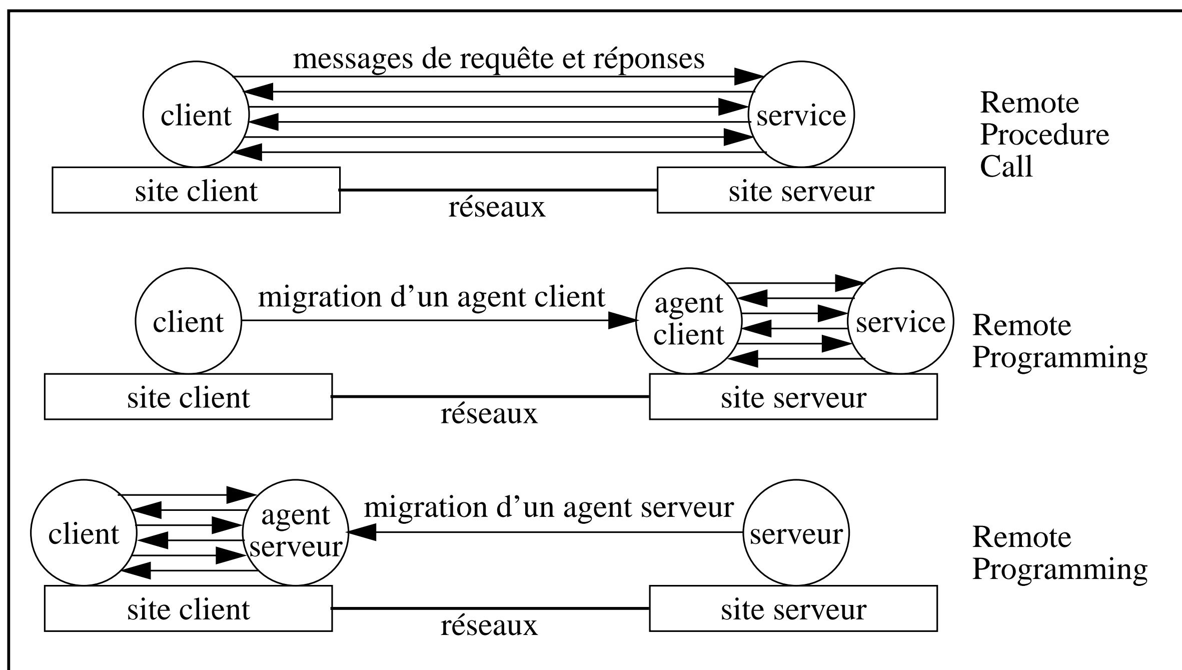


figure 14 : Le RPC et les deux modes du "Remote Programming"

L'agent est un message qui comporte des données et un comportement, définis à l'aide du langage interprété orienté objets "Telescript". Ce message est susceptible de transiter par tout type de réseau et de s'exécuter sur tout type de matériel (du *communicateur* personnel portable au *mainframe*). La migration est déclenchée de manière explicite (instruction "go") et conserve l'activité en cours. Les agents, passifs au cours de leurs transferts, sont exécutés par un

"Telescript Engine" à leur arrivée dans une *place*. Une place est un lieu d'accueil, d'exécution et d'interaction entre agents. Même lorsqu'ils se trouvent dans des places distinctes, deux agents peuvent s'échanger des objets en établissant une *connexion*.

On voit, à travers cette approche, que le nomadisme de l'agent est lié à un fort besoin d'interactivité en dépit d'une répartition à grande échelle (potentiellement le monde entier) des clients et des serveurs, et de réseaux de communication aux caractéristiques variables. Enfin, on note que la migration en environnement hétérogène et la conservation de l'activité sont rendues possibles par l'utilisation d'un langage interprété.

4.4.5 Les agents intelligents

Le besoin d'intelligence et le manque actuel affleurent dans tous ces exemples. Dans le cas de l'accès aux services et aux informations répartis, il s'agit d'optimiser la rapidité, les chances de réussite, la pertinence du résultat, voire le coût. L'administration de réseau présente des problématiques comparables, et les mécanismes pseudo-intelligents de découverte automatique de réseau ne peuvent que bénéficier d'outils adaptés au raisonnement.

Dans le cadre général de l'interface homme-machine, [BYT 94] présente les recherches d'IBM en matière de "Human-Centered Computing". Cette approche vise à adapter les systèmes informatiques au mode de communication des utilisateurs, en intégrant notamment des types de données plus riches (son, vidéo) et en élargissant les types d'interaction possibles. Pour assurer une liberté permanente d'interaction (voix, écriture manuscrite, clavier, souris, stylo optique, écran tactile…), mais aussi pour rechercher une efficacité optimale de ces interactions, le système doit apprendre à connaître l'utilisateur. Cette intelligence de la machine repose sur des *agents*, sortes de modules logiciels qui utilisent des techniques et des technologies avancées afin d'analyser, apprendre, comprendre les requêtes en langage naturel parlé ou manuscrit d'un utilisateur donné. Ainsi, pour une requête telle que "appeler ma femme", le système devra déduire qu'il s'agit de téléphoner, à un numéro qu'il recherchera lui-même, en fonction de la date et de l'heure (e.g. bureau ou domicile).

4.5 Quel enjeu pour les Systèmes Répartis à Objets ?

4.5.1 Introduction

Comme nous l'avons vu au chapitre précédent, les systèmes répartis à objets cherchent à apporter des modèles et des solutions opérationnels face à des problèmes concrets de

répartition et de communication. De leur côté, les systèmes d'agents, qu'ils soient issus d'une approche de type IAD ou de considérations plus directement liées à l'informatique et aux télécommunications, ont besoin de supports pour implémenter et observer leurs modèles.

De façon immédiate, on peut exprimer le double lien qui unit les SRO et les SMA par les deux constatations suivantes :

- un système d'objets réparti peut être vu comme un système multi-agents ;
- la prise en compte de la répartition et l'utilisation d'objets pour implémenter des agents font des SRO un support technologique pertinent vis-à-vis des SMA.

A présent, il convient d'approfondir ces intuitions et d'en tirer des conclusions.

4.5.2 Quelques plates-formes multi-agents

Les acteurs

Comme nous l'avons déjà souligné en 4.3.2, le modèle acteur est particulièrement prisé pour l'implémentation des systèmes multi-agents. A ce titre, on peut citer le projet MARSALA [FER 92], dont l'objectif est de "*définir des méthodes et de concevoir des outils permettant de spécifier et de réaliser des applications utilisant un modèle conceptuel de type "agent" et mises en oeuvre à l'aide d'un mécanisme d'exécution de type acteur*". L'adéquation de ce type de modèle est exprimé en termes d'activité autonome et de moyens de communication asynchrone. Les outils et environnements de développement sont basés sur la sélection de trois langages d'acteurs (ACTALK, extension de Smalltalk — CLAS, basé sur Common Lisp — MERING IV), qui traduisent une orientation objet, ainsi qu'un besoin de multiactivité et de répartition.

Autre langage de programmation par acteurs, PLASMA II est proposé par [ARC 93] comme candidat pour l'implémentation de systèmes multi-agents. Il enrichit le modèle de base par des primitives de communication avancées, compatibles avec celui-ci (i.e. exprimables à partir des primitives acteur décrites par Gul Agha) :

- envoi de message bloquant, avec attente de message de réponse ;
- envoi de message avec attente différée de message de retour ;
- diffusion de message à plusieurs acteurs ;
- priorités.

Kephren

[STI 93] remet en cause l'idée d'une totale adéquation du modèle acteur aux systèmes multi-agents, et présente un modèle réflexif à objets concurrents, susceptible de mieux répondre aux besoins de représentation de groupes d'agents. En permettant à un système de manipuler sa propre représentation par des opérateurs introspectifs, la réflexivité offre aux objets des capacités de contrôle des activités qui ne sont pas immédiates dans les langages d'acteurs. De plus, cette réflexivité s'exprime aussi au niveau de groupes d'objets concurrents. Le modèle Kephren a été implémenté avec le langage à objets Smalltalk, qui permet notamment une hétérogénéité matérielle par le biais de la communication entre images Smalltalk.

GTMAS

Avec GTMAS, [CHE 93] propose une boîte à outils générique pour la construction et l'évaluation de systèmes multi-agents. Les agents sont des processus s'exécutant de manière concurrente, pouvant être inactifs ou actifs s'ils sont “réveillés” par des stimuli extérieurs (i.e. des communications). Sans prétendre à l'exhaustivité, GTMAS propose un ensemble de primitives de base, qui exhibent un comportement par défaut pouvant être modifié selon les nécessités d'un domaine d'application particulier. Les principales fonctionnalités de GTMAS sont les suivantes :

- La *représentation des agents* contient un nom unique, un état interne d'activité, un corps (partie procédurale contenant son savoir-faire), une boîte aux lettres, des intérêts (filtrage des communications jugées intéressantes), une base de faits locale, et une procédure d'initialisation.
- Les *communications* se font par envoi de messages ou partage d'information dans un tableau noir. Un message peut être envoyé à un destinataire unique, à une liste de destinataires, à des destinataires vérifiant un prédicat, voire à tous les agents du système. L'acheminement des messages est supposé parfait (ni altération, ni perte de message).
- Un *moteur d'inférences* permettant de représenter la connaissance de l'agent sous forme de base de règles.

Les grandes tendances

La présentation de ces plates-formes multi-agents montre des tendances fortes :

- l'utilisation généralisée de langages à objets ;

- le besoin de gestion d’activités ;
- la communication basée soit sur l’envoi de message vers des boîtes aux lettres, soit sur une mémoire partagée (cf. tableau noir) ;
- la coopération entre des agents qui jouent à la fois un rôle de client et de serveur ;
- le besoin de structures d’organisation des agents.

La communication par message de base est asynchrone, mais elle très souvent complétée par des primitives permettant la synchronisation et la diffusion. Cette diffusion nécessite non seulement d’être restreinte pour éviter de submerger le système, mais également de prendre en compte l’évolution de la population d’agents. Le problème de l’ouverture et de l’échelle des systèmes est très rarement abordé, le protocole omniprésent dans les SMA étant celui du *réseau de contrat* (cf. [SMI 80], 4.2.3 et 8.2). Il existe néanmoins quelques approches permettant une connaissance incomplète des destinataires potentiels et une diffusion limitée, telles que le modèle du *cri* [ARC 93], ou de la *relaxation restreinte* [CAM 95]. Leur principe repose sur une transmission aux accointances des messages d’appel d’offre reçus, suivant une éventuelle évaluation de pertinence et un facteur d’atténuation (e.g. fonction de la “position” des agents, du nombre de retransmissions ou du temps écoulé depuis l’émission initiale).

4.5.3 L’offre des SRO

Face aux besoins des SMA, les SRO offrent aujourd’hui une architecture de communication orientée objets, transparente à la répartition et à l’hétérogénéité (cf. CORBA). Les mécanismes de communication offerts sont généralement synchrones (invocation de méthodes), avec quelques variations (cf. “best effort” ou “at most one”, 3.4.3), mais ils permettent de façon immédiate de construire des boîtes aux lettres et des primitives d’envoi de message. On peut alors bâtir des agents au comportement mixte de client et de serveur.

Certains systèmes, comme COOL (cf. 3.5), disposent en plus de mécanismes de communication de groupe, qui peuvent être utilisés pour gérer des organisations d’objets. En ce qui concerne les structures de tableau noir, rares sont les systèmes qui proposent une véritable mémoire partagée répartie et cohérente, mais il est possible de l’émuler par un objet serveur. Quant à la gestion d’activités, elle apparaît de façon croissante dans les langages et les systèmes, COOL étant une illustration de ce phénomène dans le domaine des SRO.

Au-dessus de l’architecture de base, les services communs de CORBA introduisent des fonctionnalités de plus en plus intéressantes (certaines réalisées, d’autres en cours de spécification), parmi lesquelles on peut citer :

- le nommage des objets (avec différents “contextes” de nommage facilitant la gestion de noms uniques malgré la répartition) ;
- la notification d’événement, mécanisme de communication particulièrement intéressant pour introduire des propriétés réactives dans les objets ;
- la migration (confrontée à deux difficultés majeures, l’une concernant l’hétérogénéité, l’autre ayant trait à la conservation de l’activité en cours) ;
- le *trader* (cf. courtier, 3.3.6), dont le rôle est de mettre un client en relation avec le serveur optimal. L’adéquation du serveur suppose une conformité d’interface et la satisfaction de contraintes exprimées sur des attributs caractérisant la manière de rendre le service requis.

A travers ce principe de satisfaction de contraintes, qui pourrait être étendu par une véritable résolution optimisant une fonction de coût, on voit que l’introduction de mécanismes avancés offre des services objets “intelligents”. Par ailleurs, les facilités communes de CORBA définies pour la *gestion de tâche* incluent une notion d’*agent*, *statique* ou mobile, exécutant un *script* (scénario d’activité décrivant de façon déclarative des objectifs à réaliser). On voit que ces agents répondent directement aux préoccupations des ingénieurs de recherche en informatique et télécommunication (cf. 4.4). Néanmoins, les mécanismes proposés pourront également intéresser la communauté IAD, d’autant plus que la spécification d’une interface de système à base de règles est en cours. Enfin, signalons que l’OMG travaille à l’attachement de sémantique aux opérations et aux requêtes.

4.5.4 Bilan

On peut tirer deux conclusions :

(1) L’approche objet et les primitives de communication des SRO permettraient aux plates-formes multi-agents de s’affranchir du redéveloppement des fonctionnalités de communication. D’ailleurs, il existe de nombreux SMA simulant la répartition et la concurrence sur une seule machine, alors que les SRO pourraient prendre en charge une répartition physique effective. De plus, en adoptant une architecture de communication répartie commune, des systèmes différents se verraient offrir une opportunité de coopération.

(2) Il existe un ensemble de problématiques communes aux SRO et aux SMA, de telle sorte que certaines recherches en IAD pourraient permettre de dégager des solutions génériques. Les problématiques liées à la répartition et à l'ouverture des systèmes peuvent s'exprimer en termes d'accointances évolutives, incomplètes, incohérentes et sur-abondantes, mais aussi d'hétérogénéité sémantique. Alors que de nombreux SMA supposent un système sous-jacent parfaitement fiable, l'IAD ne doit pas perdre de vue l'ensemble des problèmes très concrets posés par la répartition, notamment vis-à-vis de l'échelle du système.

Pour résumer, en reprenant une phrase déjà citée en introduction, *"l'intelligence au sens prise de décision a besoin des systèmes répartis comme les systèmes répartis feront appel à l'intelligence"*. Nous pensons que l'émergence du standard CORBA et de produits conformes opérationnels, ainsi que la disponibilité de services objets avancés, constituent une aubaine pour les plates-formes multi-agents[6]. Réciproquement, les systèmes répartis à objets sont susceptibles de tirer profit de contributions apportées par la communauté IAD.

D'un point de vue global, l'enjeu des SRO pourrait être de fournir un point de rencontre opérationnel entre les applications réparties et l'IAD.

4.6 Conclusion

Comme nous l'annoncions au début de ce chapitre, la notion d'agent est loin d'être figée. Utilisée par différentes communautés à des niveaux variablement conceptuels, elle n'apporte pas un modèle universel. Ceci fait des SMA une remarquable source d'inspiration qui propose des modèles, des outils, mais aussi de nombreux points d'ouverture et des pistes de recherche (éthologie, sciences sociales, systémique, IAD et techniques d'IA).

Si l'on essaie d'esquisser un tableau global des approches multi-agents, on s'aperçoit que les informaticiens ont besoin d'aller au-delà du modèle objet, même réparti. En effet, la répartition naturelle des problèmes et l'augmentation de leur échelle, due à une nécessité croissante d'ouverture des systèmes et d'interopérabilité, imposent des capacités d'observation, d'adaptation, d'autonomie, de coopération, de raisonnement. Ce besoin d'intelligence rend capital un rapprochement avec la communauté IAD, qui, pour sa part, doit prendre en considération ces problèmes nouveaux. Systèmes ouverts et dynamiques,

6. Accessoirement, on remarquera certaines fonctionnalités avancées de COOL qui, dans sa version native Chorus, tire profit des spécificités du système réparti sous-jacent.

répartition à grande échelle, interdisent certaines hypothèses généralement admises, notamment en matière de communication (fiabilité parfaite, accointances restreintes et/ou statiques et/ou exhaustives, possibilité de diffusion globale…) et requièrent de nouveaux modèles. Vis-à-vis de nos préoccupations en matière de système d'information et de bureautique communicante, les problématiques que nous retenons sont les suivantes :

- trouver l'agent serveur le plus adapté dans un environnement réparti, ouvert et dynamique ;
- négocier avec l'agent serveur pour obtenir le service le plus adapté.

Il s'agit de problèmes très généraux et très vastes, qui se posent dans tout système multi-agents ainsi que dans les systèmes répartis : des agents ou des objets réclament des services sans pouvoir parfaitement les spécifier, et sans connaître les serveurs potentiels. Nous proposons une vision multi-agents du système d'information, dont le rôle est de servir de base pour la coopération et le raisonnement nécessaires à une interopérabilité optimale. Nous aborderons ensuite les deux problématiques réduites à un système d'échelle limitée, mais nous indiquerons des pistes de généralisation visant à prendre en compte un système global plus large.

Chapitre 5

La vision multi-agents du système d'information

"The area of Computer Supported Cooperative Work provides one of the most fundamental applications for Distributed Artificial Intelligence research." [HAU 93]

5.1 Une modélisation multi-agents

5.1.1 De l'interopérabilité à la coopération

La viabilité d'un système d'information dépend de sa faculté d'intégration dynamique de ressources très diverses (machines, applications, utilisateurs) et sujettes à évolution. La variété des interactions entre ces ressources de types très différents s'ajoute à l'hétérogénéité des matériels et des logiciels pour rendre complexe l'interopérabilité. Ainsi, la complexité se révèle non seulement technique, mais aussi d'ordre sémantique, les points de vue sur l'information et sa représentation étant très variés. Cette double complexité est accrue par l'ouverture résultant du besoin d'interopérabilité entre systèmes d'information. En effet, leur interconnexion introduit un degré supplémentaire d'hétérogénéité, tant d'ordre technique que sémantique, plus difficilement maîtrisable puisque d'origine externe.

La complexité technique trouve un premier niveau de solution dans les approches de type système réparti à objets, notamment sur micro-noyau (cf. Chapitre 3). Les systèmes répartis à

objets parce qu'ils permettent de s'affranchir de l'hétérogénéité matérielle, logicielle et réseau, tout en offrant une capacité d'intégration des applications à travers un modèle objets. La technologie micro-noyau parce qu'elle propose une communication uniforme entre tout type de matériel, ainsi qu'un support pour diverses personnalités de système d'exploitation.

Cependant, la complexité issue de l'hétérogénéité des points de vue, de la dynamique et de l'ouverture des systèmes d'information, induit un besoin de modèles et d'outils de plus haut niveau. Il faut pouvoir se baser sur une sémantique partagée (notion de *principe intégrateur*, cf. Chapitre 2) et sur des capacités de raisonnement pour assurer une mise en relation optimale (délai, coût, adéquation) et une négociation fine entre les serveurs et les clients. La notion de coopération correspond à cette interopérabilité intelligente, qui consiste en une démarche active des serveurs et des clients visant une collaboration optimale.

5.1.2 Obtenir un système uniforme

Quelques abstractions

Dans le cadre plus restreint de la bureautique communicante, les difficultés d'ordre sémantique sont amoindries, du fait de l'homogénéité du contexte. On peut ainsi voir un système de bureautique communicante comme un réseau de machines (*sites*) qui disposent de certaines *ressources* susceptibles d'apparaître, disparaître ou évoluer : programmes d'application (e.g. base de données, éditeurs divers...) ou entités applicatives nomades (e.g. objet "document électronique" en circulation), périphériques (e.g. imprimantes) pilotés par des couches logicielles de niveau variable, utilisateurs connectés via un programme d'interface. A chacune de ces ressources correspond un ensemble propre de *services* disponibles auxquels sont associées certaines *caractéristiques*. Les caractéristiques de service se déclinent en deux catégories :

- Les *caractéristiques structurelles*, statiques ou faiblement dynamiques en cas de mise à niveau, sont propres à l'implémentation particulière d'un service. Par exemple, pour un service d'impression offert par une imprimante, il peut s'agir de la marque, du procédé utilisé, des formats reconnus, de la vitesse d'impression...
- Les *caractéristiques conjoncturelles*, typiquement dynamiques, dépendent du contexte d'invocation du service. En poursuivant l'exemple du service d'impression, on peut citer le nombre d'impressions en attente, l'indisponibilité pour cause de maintenance, l'indice d'usure du dispositif d'impression, le nombre de feuilles disponibles...

Chaque ressource peut avoir un rôle mixte de serveur (offre de service) et de client (recherche de service). Comme nous l'avons présenté en 5.1.1, la coopération entre ces ressources se heurte à deux difficultés :

(1) entrer en relation avec les ressources ad hoc dans un système dynamique ;

(2) une fois le contact établi, négocier de la façon la plus précise possible le service requis.

Une vision multi-agents

La grande interopérabilité requise pour satisfaire ces deux points doit tirer partie d'une intégration des ressources dans une représentation uniforme du système. Pour cela, on associe à chaque ressource un objet chargé de la représenter vis-à-vis des autres ressources. Ceci permet de créer une couche uniforme d'objets résultant de la projection des entités du système, qu'elles soient elles-même objets ou non (cf figure 15). C'est dans cette couche qu'ont lieu les recherches de serveurs et les négociations de service.

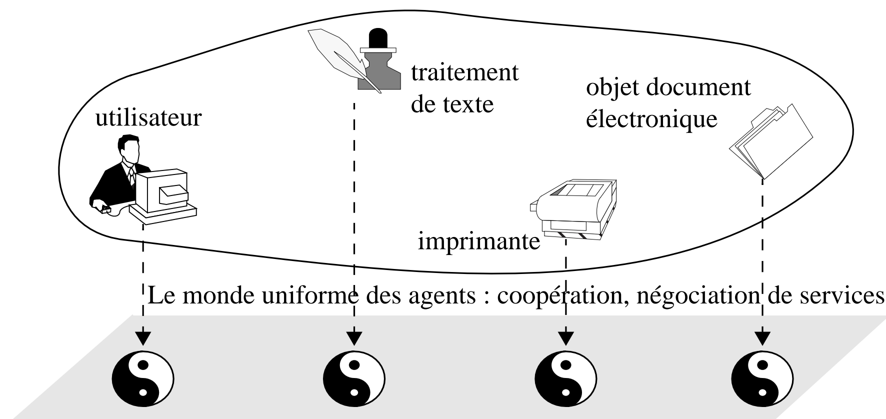


figure 15: représentation uniforme du système

Nous qualifions ces objets d'agents, dans la mesure où ils constituent des entités actives autonomes, dotées de capacités de raisonnement basées sur une représentation de l'environnement, des autres agents, et de la ressource que chacun d'entre eux représente. Chaque agent possède un rôle d'intermédiaire négociateur, soit en tant que client, soit en tant que serveur. La notion de service est généralisée, puisque toute communication n'est que requête de service[1].

1. Rien empêche de définir des services correspondant à des familles de communications typées, tels que des actes de langage (“question”, “réponse”, “information”…)

5.1.3 Optimiser la coopération

Trouver un serveur

A priori, un agent ne possède pas d'accointances lorsqu'il apparaît dans le système. Réciproquement, il est inconnu de la grande majorité des autres agents. De surcroît, les accointances d'un agent doivent évoluer en permanence du fait de la dynamique du système. Par conséquent, le besoin de coopération rend nécessaire la mise en place de structures particulières et de protocoles associés, dédiées à la mise en relation des clients et des serveurs. Le parallèle immédiat avec les systèmes répartis à objets permet d'imaginer des annuaires de services (cf. *courtier*, Chapitre 3) mais d'autres mécanismes sont possibles (e.g. analogie avec la publicité, les petites annonces, les pages jaunes...). A sa création, tout agent doit être capable de contacter et d'utiliser une telle structure. Au cours de l'évolution du système, les agents doivent être capables de prendre en compte l'apparition, la modification, ou la disparition de structures et de protocoles.

Négocier le service

Dans notre approche, toute communication entre agents n'est qu'invocation de service. Il est donc nécessaire qu'une certaine sémantique soit partagée pour décrire ces services. Comme nous l'avons déjà précisé, la restriction à un contexte de bureautique communicante simplifie la définition d'une telle sémantique. Un service est donc simplement représenté par un symbole, connu par les agents susceptibles de l'invoquer, et dont la signification est intrinsèque. Toutefois, cela n'exclut pas une vision macroscopique, un service invoqué pouvant être décomposé en plusieurs service, de façon transparente ou explicite. Ainsi, un service de rédaction de fax peut mettre en œuvre un service d'édition (traitement de texte), puis un service de conversion de format électronique, et enfin un service de transmission par modem.

Classiquement, l'invocation d'un service impose la spécification d'un certain nombre d'arguments (cf. méthodes d'un objet). Comme nous envisageons une négociation fine, il s'agit ici de fixer un certain nombre de contraintes sur les caractéristiques du service. La négociation consiste alors à vérifier la compatibilité entre les contraintes exigées par le client et celles imposées par le serveur, puis à rechercher la solution qui satisfait le plus le client. Ce dernier point est rendu par une fonction de satisfaction fournie par le client, et que le serveur tente de maximiser en ajustant la valeur des caractéristiques du service non encore figées par les contraintes. Si, malgré ces mécanismes, la valeur de certaines caractéristiques reste

indéterminée, le serveur possède lui aussi des critères lui permettant de fixer des valeurs par défaut. Il peut s'agir d'une fonction de coût à minimiser, mais aussi de simples règles, établissant, par exemple, que le support par défaut pour une impression est une feuille de papier blanc au format A4.

Gérer les niveaux de sémantique

Le contexte homogène de la bureautique communicante n'empêche pas les différences de niveau sémantique entre agents, et particulièrement entre les clients et les prestataires d'un service donné. En effet, les serveurs sont des spécialistes dans leur domaine, et ils connaissent précisément les caractéristiques et leur signification pour chaque service qu'ils offrent. En revanche, la connaissance du point de vue du client peut être plus ou moins approximative. Il existe cependant un niveau zéro de sémantique constitué de trois racines :

- le coût,
- le délai,
- la qualité du résultat.

Au-delà de ce niveau, des raffinements successifs de la sémantique dans le cadre d'un service donné, permettent de définir une arborescence de caractéristiques *floues* (ou qualitatives). Les feuilles de cet arbre sont les caractéristiques *terminales* (ou quantitatives), qui sont les seules valuables par le serveur (figure 16).

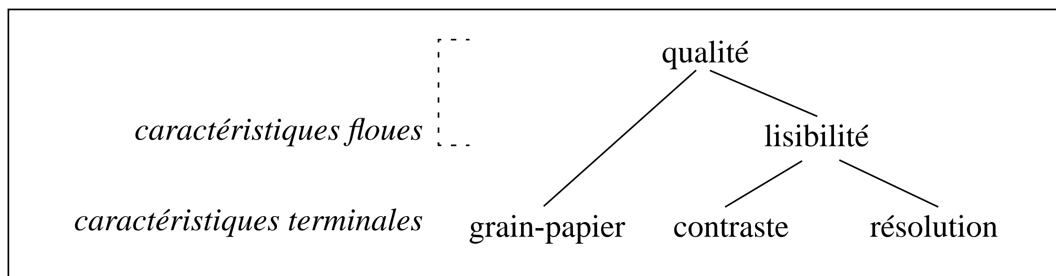


figure 16 : exemple de niveaux sémantiques pour un service d'impression

Les contraintes elles-mêmes peuvent être *qualitatives* ou *quantitatives*. Dans le cas d'une caractéristique floue, la contrainte est nécessairement qualitative. En revanche, une contrainte portant sur une caractéristique terminale peut être qualitative ou quantitative. Ainsi, dans l'exemple de la figure 16, ni la qualité, ni la lisibilité ne peuvent être valuées. Le client demandera donc une lisibilité prioritaire, bonne, moyenne ou indifférente (voire mauvaise ?). Par contre, la résolution pourra être "bonne", ou plus précisément supérieure à 400 dpi.

Le serveur est capable de convertir tous ces types de contraintes pour parvenir à des caractéristiques terminales dont il va rechercher la valuation. Celle-ci pourra être transmise au client, pour simple information ou en vue de fournir des données pour d'éventuelles capacités d'apprentissage (e.g. analyse des caractéristiques et du résultat obtenu pour plusieurs invocations d'un même service).

5.2 Discussion

5.2.1 L'approche agents

Il s'avère que notre approche multi-agents de la bureautique communicante n'est pas un cas unique. En effet, le projet IMAGINE présente une approche tout-à-fait comparable dans le domaine du "Computer Supported Cooperative Work". [HAU 93] explique que les développements en matière de CSCW ont conduit à des applications qui facilitent le travail coopératif entre utilisateurs, mais qui s'intégrent très mal les unes aux autres. D'où la proposition d'un rapprochement vers l'intelligence artificielle distribuée, qui propose des modèles anthropomorphiques de coopération, initialement destinés à la collaboration d'entités logicielles. En combinant les techniques issues de l'IAD et du CSCW, le projet IMAGINE se concentre sur la notion de "Human-Computer Cooperative Work", qui exprime la capacité de coopération indifférente entre humains et machines. Un modèle multi-agents est mis en œuvre, et l'utilisation du terme agent est justifié par les caractéristiques suivantes :

- entité autonome,
- communication et coopération,
- résolution de problème, potentiellement collective.

Le parallèle avec notre projection multi-agents du système d'information est donc immédiat. En effet, le principe de notre couche homogène d'agents coopérants, représentant indifféremment les machines, les applications et les utilisateurs, relève directement du HCCW. Les agents représentant un utilisateur peuvent d'ailleurs constituer la partie intelligente de l'interface homme-machine, point d'accès à tous les services disponibles à travers les réseaux. De façon similaire aux agents IMAGINE, nos agents présentent des capacités de résolution de problème (décomposition de but en sous-buts, génération de plan, résolution de contraintes).

Toutefois, même si notre approche est moins aboutie que ce projet Esprit, elle présente la particularité de s'intégrer techniquement à un Système Réparti à Objets. Ceci permet d'aller

plus loin dans la décomposition multi-agents (les objets manipulés par les applications peuvent être vus comme des agents), de bénéficier d'un support de communication puissant, et de prendre en compte la répartition plus finement. En outre, nous effectuons un rapprochement complémentaire (SRO-IAD), source d'outils, de modèles et de points de recherche génériques.

5.2.2 A propos des difficultés de coopération

Des structures et des protocoles

Notre principe d'intégration par une couche d'agents offre une base de coopération mais ne résout pas les problèmes de façon immédiate. En effet, l'ouverture et la dynamique des systèmes considérés font qu'il est très difficile pour un agent de connaître tous les services et, plus encore, les serveurs disponibles. Ce besoin d'accéder à des serveurs inconnus a priori est un problème général dans les systèmes répartis à objets[2].

Dans l'OMG, le nommage symbolique d'une interface d'un objet serveur offre un premier niveau de service en la matière, puisqu'il permet d'associer un nom donné à une référence non déterministe. Mais cette association est insuffisante, car elle impose une connaissance du type d'interface et la capacité de “calculer” le nom d'une instance. Le principe du courtier (voir Chapitre 3), entité permettant de trouver un serveur quelconque exportant une opération donnée, illustre le besoin de structures de coopération et de protocoles associés.

La négociation

Lorsqu'un agent recherche un service, la simple spécification d'une opération et de la valeur des arguments est insuffisante. En effet, le client peut se satisfaire d'une gamme étendue de variations du service demandé, pourvu que certaines caractéristiques soient respectées. Dans le domaine du multimedia réparti, mais également dans les systèmes répartis à objets, la notion de contrainte s'impose désormais, respectivement en termes de “Qualité de Service”, ou de contraintes élémentaires sur les attributs d'une opération pour les courtiers.

L'optimalité de la recherche et de la négociation d'un service est donc également un problème général dans les systèmes répartis. Notre principe de contraintes sur des caractéristiques permet une négociation optimale, mais se confronte aux limitations des techniques de résolution de contraintes.

---

2. Par exemple, l'insuffisance des services fournis par le système a conduit à la création d'une couche d'objets répartis facilitant la coopération dans CIDRE (voir “Gestionnaires d'Objets”, Chapitre 2).

5.3 Vers des services "intelligents"

5.3.1 CIDRE

La problématique du "Workflow"

L'application répartie CIDRE est à l'origine de nos travaux en matière de besoin de raisonnement (cf. Chapitre 2). En effet, la Circulation Intelligente de Dossiers REpartis n'exprime de l'intelligence qu'au moment de la création d'une procédure bureautique. A partir d'une base de faits décrivant la procédure, un système expert engendre un schéma de circulation, après avoir détecté les incohérences et éliminé les ambiguïtés en dialoguant avec l'administrateur CIDRE.

Mais à la création d'une instance du dossier correspondant, et pendant l'exécution de la procédure, aucun raisonnement n'est mis en œuvre. Or, le choix d'une action à déclencher et d'un exécutant nécessite une adaptation au contexte de circulation (configuration globale du système, historique du dossier). Par exemple, le choix d'un intervenant, représenté dans la procédure par un *rôle bureautique*, est un problème délicat, qu'un simple annuaire de service ne résout pas. Absences temporaires ou prolongées, compétences, délégations à géométrie variable, responsabilités transversales doivent être prises en compte différemment suivant qu'un dossier particulier privilégie la sécurité, la rapidité ou le coût. De plus, des traitements d'exception doivent être envisagés vis-à-vis du schéma de circulation si celui-ci s'avère inadapté au contexte.

Nous voyons donc clairement que l'intelligence centralisée et exercée a priori ne suffit pas. Un raisonnement réparti entre les entités impliquées dans une procédure, au moment où elle se déroule, est indispensable à la viabilité d'une application de type "Workflow" au sein d'un système un tant soit peu ouvert et dynamique.

La démarche multi-agents

Avec la représentation multi-agents de l'univers bureautique, nous avons la possibilité d'introduire une grande quantité d'information concernant les différentes ressources du système. La répartition de ces agents et leur représentation exclusive des ressources en font des sources cohérentes de données, à l'accès homogène.

Dans le cas de CIDRE, la représentation de chaque utilisateur par un agent permet de disposer d'informations sur ses compétences et attributions, les projets dans lesquels il est

impliqué, ses disponibilités… Un agent dossier en cours de circulation peut alors raisonner sur les caractéristiques de plusieurs agents utilisateurs pour choisir le plus adapté à la situation, choisir parmi une liste d'actions possibles, ou bien encore déclencher une exception de circulation.

La représentation uniforme de chaque ressource par un agent permet également à CIDRE de gérer des Workflow au sens général du terme, en interagissant de la même manière avec des machines (e.g. envoyer un fax, imprimer un document) ou des applications (e.g. accès à une base de données). Toutefois, ne perdons pas de vue que la recherche des agents serveurs nécessite la mise en œuvre de structures et de protocoles de coopération adaptés.

5.3.2 Messagerie intelligente

"la voiture immatriculée 314 PI 14 a ses feux allumés". Qui n'a pas reçu — et détruit — de tels messages électroniques, diffusés à l'ensemble d'une organisation locale ? Pourtant, ce message n'intéresse qu'une seule personne, et l'ensemble des autres utilisateurs doivent le consulter avant de le détruire. Un premier niveau d'intelligence concerne la boîte aux lettres de chaque utilisateur, qui peut faire le tri des messages non pertinents. Le problème réside alors dans l'analyse du sujet et du corps du message, dont le contenu n'est pas structuré. Cette technique se confronte au dilemme classique suivant :

- ne pas supprimer des messages pertinents,
- exercer un filtrage suffisamment efficace.

Ici, comme dans d'autres cas similaires (e.g. recherche d'information sur un sujet précis), il paraît alors plus élégant de n'émettre que le (ou les) message(s) nécessaire(s). En effet, seul l'émetteur peut définir le type de destinataire qu'il souhaite atteindre, mais un nom ou une liste de diffusion ne sont pas adaptés dans ce cas. D'où l'idée d'un service de messagerie intelligente où le destinataire est spécifié suivant le principe des contraintes sur les caractéristiques des agents destinataires. Là encore, ce type de fonctionnalité avancée nécessite des structures de coopération afin d'orienter les recherches. Par exemple, on peut envisager un service de type annuaire, permettant de faire un premier tri sur des caractéristiques structurelles.

Mais l'approche multi-agents permet d'aller au-delà, avec la notion de message intelligent. Que ce soit en tant qu'agent message ou qu'il s'agisse d'un message contenant un script, un message exécutable est capable d'effectuer lui-même un niveau de tri

supplémentaire. En effet, il peut interagir avec l'agent représentant l'utilisateur destinataire pour déterminer la pertinence de sa présence dans la boîte aux lettres. Ce message doté d'un comportement peut alors déclencher de nombreuses actions : auto-destruction, renvoi vers un autre destinataire, réponse automatique, réflexes si le délai de lecture est trop long… A travers ces fonctionnalités avancées apparaissent deux caractéristiques intéressantes :

- le message “actif” peut contenir ses propres protocoles de communication,
- il peut chercher à s'adapter à un agent “ancien”, voire le mettre à jour.

5.4 Bilan

L'approche multi-agents des systèmes de bureautique communicante se révèle tout à fait pertinente. Elle offre un modèle simple permettant d'intégrer des ressources hétérogènes (applications, machines et utilisateurs) dans une couche d'agents coopérants. Ces agents sont à la fois clients et prestataires de services, et négocient sur un principe de résolution de contraintes. Les notions de contraintes qualitatives et de caractéristiques floues permettent une négociation fine en dépit d'une connaissance potentiellement grossière de la part du client.

Les agents, interlocuteurs exclusifs pour l'accès aux ressources, constituent des sources d'informations cohérentes mais réparties dans l'ensemble du système considéré. La répartition, l'ouverture et la dynamique de ce système impliquent que des agents intermédiaires proposent des services d'aide à la recherche de serveurs. Ce besoin de structures et de protocoles de coopération est un problème général, qui concerne également les systèmes répartis à objets. Ainsi, nos travaux consistent bien en un triple rapprochement :

- le domaine d'application de la bureautique communicante ;
- le support technique des système répartis à objets ;
- l'apport théorique de l'intelligence artificielle distribuée.

La partie qui suit illustre cette approche par une implémentation partielle et des pistes de recherche pour les aspects non traités.

PARTIE 2

OUTILS POUR LA MODÉLISATION MULTI-AGENTS ET MISE EN ŒUVRE

Introduction

Dans cette partie, nous nous proposons tout d'abord de présenter les fonctionnalités et l'implémentation de PUMA, une plate-forme destinée à la mise en œuvre de systèmes d'IAD. Intégré à COOL, cet environnement de développement permet de pousser l'approche multi-agents au-delà d'une simple répartition logique de l'architecture logicielle, en ajoutant des mécanismes de répartition physique des agents.

Ensuite, nous décrirons une application de PUMA à notre approche multi-agents du système d'information de l'entreprise, dans un cas concret de service lié aux activités collectives. Puis, à la lumière de cette réalisation, nous mettrons en évidence le besoin de structures et de protocoles de coopération "intelligents", adaptés au type de service demandé et aux contraintes de l'environnement.

Enfin, nous conclurons la thèse...

Prolog pour Univers Multi-Agents ? L'utilisation de C++ dans le cadre de COOL se justifie par son orientation objets et son efficacité issue de C, langage traditionnel des systèmes UNIX. Mais il se révèle limité ou inadapté à certains de nos besoins (représentation des connaissances, langage de haut niveau pour la négociation, comportements "intelligents" et adaptatifs). Comme l'exprime [STI 94], les outils du type C++ peuvent être considérés comme des langages de niveau assembleur du point de vue des objets et de la répartition. Le rapprochement effectué en direction de l'Intelligence Artificielle Distribuée, suite à la modélisation multi-agents, conduit à s'intéresser à d'autres langages couramment mis en œuvre dans ce domaine. Pourtant, on ne peut se passer des fonctionnalités offertes par COOL, d'où l'idée d'intégrer un langage de ce type dans un objet C++ sur COOL.

Le choix de Prolog découle de plusieurs considérations (cf. [DIL 95b]). D'un point de vue général, il s'agit d'un langage interprété, de haut niveau, disposant d'une représentation uniforme des données et des programmes. Par ajout de prédicats de communication, les programmes peuvent alors non seulement échanger du savoir mais aussi du savoir-faire. Dans

le cadre de notre modélisation, cela permet de propager des procédures nouvelles, ou des modifications d'anciennes. En effet, puisqu'un programme Prolog a accès à son propre code et peut se modifier, les agents peuvent se mettre à jour dynamiquement, sans interruption de leur fonctionnement. En outre, ce mécanisme fonctionne également en milieu hétérogène.

De plus, Prolog fournit une base de langage de communication commune à tous les agents. D'ailleurs, son utilisation pour définir et interpréter des langages spécifiques à nos besoins de coopération est à rapprocher de ses fonctions initiales d'analyse du langage. Prolog peut être notamment utile pour notre langage de négociation qui est basé sur la résolution de contraintes. A cet égard, l'adéquation de l'association Prolog/contraintes n'est plus à démontrer, puisque de nombreux systèmes de programmation par contraintes s'appuient sur des extensions de la programmation logique (Prolog III et IV, Interlog, CHIP, IF/Prolog…).

Enfin, d'un point de vue général, les agents ne peuvent que profiter de la puissance de la résolution de but Prolog, basée sur l'unification, pour conduire des raisonnements logiques plus ou moins élaborés. Par exemple, la plate-forme et les outils développés dans le cadre du projet IMAGINE (voir Chapitre 5) sont basés sur un "langage d'interaction multi-agents" (MAIL) intégrant IC-Prolog][.

Quel noyau Prolog ? Il s'avère que le mélange C++/Prolog, traduisant une volonté d'utiliser l'outil adapté à chaque besoin particulier, ne constitue pas un cas isolé. Ainsi, dans le domaine de la configuration des ordinateurs, le besoin de support de raisonnement a récemment conduit à une intégration similaire au sein de Windows NT ([HOV 95]). Ce système d'exploitation intégre un noyau Prolog servant à configurer la carte réseau de façon automatique ("plug-and-play"). Le noyau Prolog est volontairement minimal, sans doute en partie pour des raisons d'encombrement, mais surtout parce que seuls les mécanismes fondamentaux de Prolog étaient nécessaires. Cette démarche consistant à se limiter aux bases de la programmation logique n'est pas sans rapport avec l'éloge de la *force brutale* ("brute force") faite par [BAR 95] dans le cadre des applications industrielles de Prolog.

En ce qui nous concerne, nous utilisons une implémentation très standard (de type Edimbourg) et relativement ancienne de Prolog. Parmi les raisons de ce choix, on retrouve en partie les préoccupations précitées, mais aussi la volonté de montrer la faisabilité et l'intérêt de notre approche technique. L'objectif de nos travaux n'est pas de fournir l'"outil idéal", mais plutôt de guider le choix (voire la définition) du langage le plus adapté, ainsi que les évolutions de COOL, par une expérience concrète.

Chapitre 6

PUMA, un environnement d'implémentation pour l'IAD

6.1 La classe de base PUMA

6.1.1 Présentation de l'objet Puma

De même que nous avons présenté les objets COOL comme une extension du modèle objet "standard", les objets PUMA peuvent être vus comme une extension des objets COOL qui consiste en l'intégration d'un langage interprété de haut niveau s'interfaçant avec COOL.

Le système PUMA repose sur la classe `puma` dont toute instance est un objet COOL qui comporte un interpréteur Prolog privé. Le langage Prolog intégré est enrichi de nouveaux prédicats qui encapsulent la plupart des fonctionnalités de COOL, notamment en ce qui concerne la communication (dont la migration). L'interface de l'objet puma contient les méthodes nécessaires au contrôle de ce "noyau" Prolog.

A sa création, un objet PUMA est caractérisé par deux paramètres passés au constructeur de la classe `puma` :

- le nom symbolique sous lequel l'objet doit être enregistré auprès du service de nommage COOL ;
- le nom du fichier contenant la base Prolog initiale à charger (facultatif).

Le nom associé à chaque objet PUMA est d'une importance capitale puisque tous les prédicats de communication l'utilisent pour spécifier le destinataire. En conséquence, il convient d'avoir une politique cohérente pour le choix des noms afin qu'ils soient uniques dans le domaine COOL. De part l'implémentation particulière du service de nommage réparti, l'existence de plusieurs occurrences d'un même nom entraînerait un non-déterminisme de l'objet réellement contacté.

En ce qui concerne le fichier Prolog chargé, il peut faire plus que la simple initialisation de la base du noyau Prolog. En effet, un réflexe Prolog de création peut être automatiquement déclenché en définissant une procédure puma/0 qui sera appelée dès la fin du chargement.

En dehors de cet éventuel réflexe, l'objet puma ne possède pas d'activité propre (dans la terminologie COOL, il s'agit d'un objet serveur passif). Il ne peut qu'être utilisé par d'autres objets COOL en tant que serveur Prolog via les méthodes de son interface. Dans ce cas, il est nécessaire d'assurer l'exclusion mutuelle sur le noyau Prolog qui ne supporte pas les invocations multiples.

6.1.2 Le dialecte Prolog enrichi

Conçu à l'origine comme un interpréteur Prolog enrichi de prédicats de communication (cf [TRO 92]), PUMA propose un dialecte Prolog étendu permettant d'accéder à de nombreuses fonctionnalités de COOL. Notons que les prédicats ajoutés ne réussissent qu'une fois au plus (backtrack inopérant). Les fonctionnalités introduites dans Prolog sont présentées dans ce qui suit et illustrées par la figure 17. Pour plus de détails, on consultera le tableau 18 qui décrit succinctement les principaux prédicats PUMA, ou la documentation complète de PUMA en Annexe A.

Les communications asynchrones, le service de nommage et les groupes

Les prédicats de communication asynchrones permettent d'envoyer des messages à un objet ou à un groupe en le désignant par son nom. Un objet PUMA possède un nom initial donné à sa création mais il peut le récupérer, le changer et même devenir "invisible" en se retirant du service de nommage. Les prédicats de gestion des groupes permettent à un objet de créer des groupes, s'inscrire dans des groupes ou s'en retirer. Lorsqu'un message est envoyé à un groupe, il peut être diffusé à tous ses membres ("broadcast mode") ou à un seul membre aléatoire ("functionnal mode").

```
exemple :-
    myname(N),                              % Quel est mon nom ?
    vanish,                                 % Je disparais du service de nommage
    write('je m''appelais '), write(N), nl, % (je n'ai donc plus de nom).
    appear(toto),                           % Je prends un nouveau nom.
    available(titi),                        % Y a-t-il un objet qui s'appelle 'titi' ?
    send(titi, message("toto arrive")),     % J'envoie un message à 'titi'.
    receive(groupe(G)),                     % Je consulte ma boîte aux lettres.
    addtogroup(G),                          % Je m'insère dans un groupe.
    broadcast(G, "bonjour à tous")          % Je diffuse un message sur ce groupe.
    functmode(G, "salut inconnu")           % Je contacte un membre quelconque.
    migrate(titi),                          % Je migre auprès de l'objet 'titi'.
    ringup(titi),                           % Je demande une connexion sur 'titi'.
    phone(bagof(X, je_connais(X)),R)        % Je dérive un but de façon synchrone
    write('titi connait : '), write(R),     % dans le noyau Prolog de 'titi'.
    create(agent, titi2, agent_titi).       % Je crée un agent de nom 'titi2'.

exemple :-
    interrupt("échec exemple"),             % invocation synchrone du niveau C++
    ireturn([Classe, Nom, Base_Prolog]),    % valeur de retour
    create(Classe, Nom, Base_Prolog).       % création d'un objet PUMA
```

figure 17 : invocations Prolog -> COOL/C++

Les communications synchrones

Une communication synchrone permet au programme Prolog d'un objet PUMA d'accéder directement au noyau Prolog d'un autre objet PUMA local[1]. Pour cela, une phase préliminaire de demande de connexion est nécessaire : l'objet invoqué - désigné par son nom - reçoit un message système particulier contenant notamment l'identité de l'objet appelant qui, lui, reste bloqué en attente d'une réponse (éventuellement avec un délai maximum). En cas de succès, un prédicat permet à l'objet appelant de dériver des buts[2] dans le noyau Prolog de l'autre objet, l'exclusion mutuelle étant assurée de façon transparente.

La migration

La migration d'un objet PUMA doit se faire obligatoirement au niveau Prolog par les prédicats dédiés. En effet, une migration déclenchée au niveau C++ par appel direct à la méthode COOL correspondante ne serait pas supportée par le noyau Prolog. En revanche, non seulement la migration déclenchée au niveau Prolog se déroule parfaitement, mais elle présente l'avantage de conserver l'activité Prolog en cours.

1. Ce mécanisme utilise l'invocation de méthode interobjet de COOL et n'est donc possible qu'entre deux objets PUMA partageant le même contexte.
2. Sans prise en compte du backtrack, comme pour tous les prédicats ajoutés.

prédicat	description sommaire
accept	accepte la dernière requête de communication synchrone reçue
addtogroup(+G)	ajoute l’objet PUMA dans le groupe de nom G
appear(+N)	enregistre l’objet PUMA sous le nom N
available(+N)	N est le nom d’un objet COOL dans le service de nommage
broadcast(+G, +M)	diffuse le terme M à tous les membres du groupe G
create(+C, +N, +F)	crée un objet COOL de classe C, avec les arguments N et F passés au constructeur (nom et fichier Prolog dans le cas d’une classe PUMA).
functmode(+G, +M)	envoie le terme M à un membre quelconque du groupe G
migrate(+N)	migre vers l’objet COOL de nom N
mygroups(?G)	unifie G avec la liste des groupes auxquels l’objet PUMA appartient
myname(?N)	unifie N avec le nom de l’objet PUMA
phone(+B)	dérive le but B dans le noyau Prolog de l’objet PUMA ayant accepté la dernière requête de communication synchrone.
quitgroup(+G)	retire l’objet PUMA du groupe de nom G
receive(?M)	unifie M avec le message suivant dans la boîte aux lettres
refuse	rejette la dernière requête d’invocation synchrone reçue
ringup(+N)	envoie une requête de communication synchrone à l’objet PUMA N
send(+N, +M)	envoie le terme M à l’objet COOL de nom N
vanish	retire l’objet PUMA du service de nommage

tableau 18 : les principaux prédicats “COOL” de Puma

En effet, la migration d’activité est un problème complexe qui n’a pas encore trouvé de solution satisfaisante dans le cadre des langages compilés, notamment du point de vue du coût en calcul (e.g. problème de la migration de la pile des invocations). Avec COOL, l’activité est stoppée en cas de migration puis redémarrée à zéro[3] dans le contexte de réinstallation. A charge au programmeur de maintenir certains attributs témoignant de l’état de l’activité afin d’orienter le redémarrage.

Dans le cas de PUMA, la migration est transparente vis-à-vis de la résolution Prolog en cours qui se poursuit. Cela provient du fait que la migration utilise le mécanisme de sauvegarde/restauration de l’état Prolog, qui inclut les structures de gestion de l’exécution.

3.i.e. la méthode `main()`, qui, rappelons-le, définit l’activité de l’objet, est relancée.

Bien entendu, les communications synchrones éventuellement engagées sont rompues par la migration puisqu'elles ne fonctionnent qu'en local.

La création dynamique d'objets

Tout objet PUMA peut créer dynamiquement un autre objet PUMA ou, plus généralement, un objet COOL. Ce prédicat prend en paramètre le nom de la classe de l'objet à créer ainsi que deux arguments destinés au constructeur, i.e. le nom et le fichier contant la base Prolog initiale dans le cas d'un objet PUMA.

6.1.3 L'interface de contrôle C++

L'autre point de vue consiste à considérer de façon plus immédiate qu'un objet PUMA permet d'intégrer un noyau Prolog dans un objet C++. Nous avons tenté de rendre le contrôle de ce noyau aussi simple, puissant et transparent que possible. Cela conduit à la définition de plusieurs catégories de méthodes qui nous présentons dans ce qui suit. Le tableau 19 résume les principales méthodes, et la documentation complète de PUMA, jointe en Annexe A, apportera toute précision souhaitée.

`méthode`[a]	description sommaire
`puma_call(char* but)`	dérive un `but` Prolog sous forme de chaîne de caractères
`puma_interp()`	interpréteur PUMA interactif
`puma_P()`	acquisition du sémaphore d'accès au noyau Prolog
`puma_recall()`	force un échec pour déclencher un backtrack
`puma_unif(char* var, char* val)`	récupère la valeur `val` de la variable Prolog `var` suite à une dérivation réussie (cf. résultat de l'unification).
`puma_V()`	libération du sémaphore d'accès au noyau Prolog

tableau 19 : les principales méthodes de l'interface des objets PUMA

a. Précisons pour les éventuels lecteurs non familiers des langages C ou C++ que "`char*`" désigne le type de l'argument, utilisé ici pour partager (i.e. paramètre d'entrée et/ou de sortie) une chaîne de caractères.

Le contrôle du noyau Prolog

En premier lieu, une méthode permet d'installer le noyau Prolog ou de le réinstaller après migration. Inversement, la destruction propre du noyau (libération de la mémoire, retrait des groupes et du service de nommage) est prise en charge par une autre méthode.

En dehors de ces méthodes d'initialisation/destruction, les méthodes de contrôle concernent la gestion du sémaphore du noyau Prolog. L'exclusion mutuelle est primordiale pour deux raisons :

(1) le noyau Prolog ne supporte pas les accès multiples simultanés ;

(2) chaque nouvelle résolution de but entraîne l'écrasement de l'état de la résolution Prolog en cours. En d'autres termes, l'interpréteur revient au "top level" sans possibilité de sauvegarde de l'état. Ainsi, comme le montre la figure 20, il peut être nécessaire de réserver le noyau Prolog pour une suite d'invocations.

L'invocation du noyau Prolog

Le noyau Prolog n'est réellement invoqué que par les appels `puma_call()`, `puma_recall()` et `puma_interp()`. Le premier sert à dériver un but, transmis sous forme d'une chaîne de caractères. Le deuxième permet de déclencher un échec pour rechercher une autre solution, suite à une dérivation réussie ("backtrack"). Quant au troisième, il exécute un interpréteur interactif qui se comporte comme un Prolog classique, mais au dialecte étendu et intégré à l'environnement COOL.

Le résultat de l'unification peut être récupéré sous forme d'une chaîne de caractères, soit directement au retour de l'appel `puma_call()` ou `puma_recall()`, soit par la méthode `puma_unif()`. Suivant la méthode choisie, cette chaîne contient le terme Prolog associé à une variable précise ou une liste de couples variable-valeur. A titre d'illustration, le programme de la figure 20 présume que le noyau Prolog contient des faits du type "pere(X,Y)". Il invoque une première fois le but "pere(X,_)"[4], puis procède par "backtracks" successifs jusqu'à épuisement des solutions.

```
puma_init();                        // installation du noyau Prolog
puma_P();                           // réquisition du sémaphore
etat = puma_call("pere(X, _)");     // dérivation d'un but
while (etat == POK) {
     puma_unif("X", resultat);      // résultat de l'unification
     printf("%s\n", resultat);      // et affichage.
     etat = puma_recall();          // recherche d'une autre solution
}
puma_V();                           // libération du sémaphore
puma_exit();                        // destruction du noyau Prolog
```

figure 20 : invocation C++ -> Prolog

4. Précisons, pour les éventuels lecteurs non familiers du langage Prolog, que le caractère '_' représente une *variable muette*, i.e. représentant une valeur quelconque que l'on ne désire pas connaître.

Notion d'"état PUMA"

L'utilisation de ces méthodes d'invocation nécessite certaines connaissances sur l'état interne de Prolog et, plus précisément, sur l'état PUMA. En effet, bien que les méthodes d'invocation permettent de masquer la "mécanique" interne de PUMA (cf 6.4.1 et figure 29), certaines informations doivent être remontées au programmeur pour qu'il sache si l'interpréteur Prolog est en échec, en succès définitif ou en succès avec proposition de valeur pour les variables du but dérivé. Les différents états visibles (cf figure 21), constituent un sous-

POK	succès de la résolution d'un but avec variables
PYES	succès de la résolution d'un but clos
PFAIL	échec
PERROR	erreur Prolog (but incorrect...)
PEND	session Prolog terminée (le noyau Prolog a été détruit)

figure 21 : les états visibles du noyau Prolog

ensemble des états internes, nécessaire et suffisant au pilotage du noyau Prolog. La valeur de l'état courant est retournée par les méthodes d'invocation mais elle est également stockée dans un attribut de l'objet.

6.1.4 Interopérabilité Prolog-C++

Généralités

Afin de tirer le meilleur parti de ces objets bicéphales, le programmeur PUMA doit avoir à cœur de bien déterminer et séparer les traitements et les données qui seront attribués à chacune des deux parties. Cette étape de la conception ne peut qu'être favorisée par le fait que ces deux langages sont globalement adaptés à des types de tâches distincts. Il reste ensuite à assurer un bon niveau de coopération entre les deux langages, ce qui est rendu possible par les prédicats et les méthodes présentées en figure 22. Ces mécanismes permettent de concevoir des objets composites dont l'activité est un programme C++ faisant des requêtes Prolog, un programme Prolog invoquant des méthodes C++ ou bien une combinaison des deux.

Cas particulier du prédicat interrupt/1

Le prédicat interrupt/1 permet au noyau Prolog d'un objet PUMA d'invoquer directement le niveau C++. Le traitement associé à cette requête est pris en charge par la méthode `interrupt_handler()` de l'objet. Cette méthode est définie par défaut au niveau de la

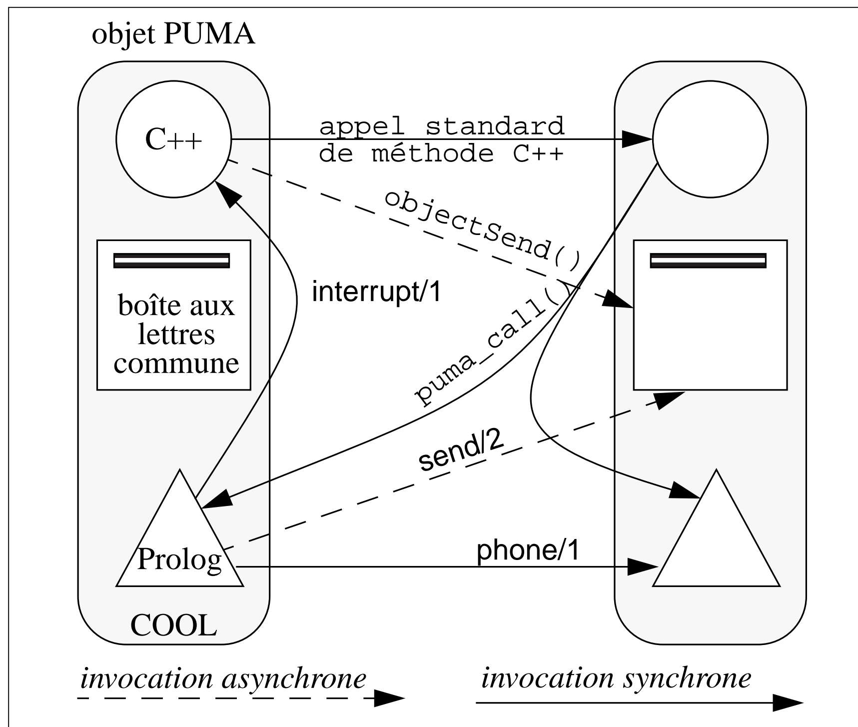


figure 22 : interopérabilité Prolog <-> C++

classe `puma` mais doit bien entendu être redéfinie par le programmeur selon ses besoins, lorsqu'il définit sa propre classe héritant de `puma`.

L'exemple de la figure 23 montre que le principe de fonctionnement est à la fois ouvert et transparent. Il permet de faire échouer ou réussir le prédicat interrupt/1 via la valeur de retour

```
int ma_classe_PUMA::interrupt_handler(char *code, char *retour) {
     switch(atoi(code)) {
     ...
     case 1 : ...
          strcpy(retour, "un(1)");  // récupérable par ireturn/1
          return(1);                // interrupt/1 réussit
     ...
     default :
          return(0);                // interrupt/1 échoue
     }
}
```

figure 23 : exemple de traitement de interrupt/1

de la méthode (vraie ou fausse au sens de C, i.e. nulle ou non nulle). En cas de succès, il est possible de retourner un terme à Prolog (sous forme d'une chaîne de caractères), récupérable par unification via le prédicat ireturn/1. On remarquera :

- l'impossibilité de prendre un compte le backtrack ;
- la possibilité d'implémenter tous les "prédicats COOL" par ce mécanisme.

Le cas échéant, le traitement assuré par cette méthode peut invoquer à son tour tout autre objet C++/COOL. Le prédicat interrupt/1 permet donc d'invoquer — indirectement — une méthode sur un objet COOL quelconque. Cependant, ce type d'invocation ne peut pas être dynamique, c'est-à-dire que pour chaque méthode invoquée doit figurer un appel "en dur" dans le code du traitement C++. Cela est principalement dû à la constitution de COOL v1 qui ne propose pas les invocations dynamiques (i.e. sans édition de lien préalable).

6.1.5 Discussion

La cohabitation de deux langages très différents, dans un environnement réparti de surcroît, est à la fois intéressante et problématique. L'intérêt réside principalement dans la complémentarité de ces deux langages que tout oppose : l'un est compilé, fonctionnel, orienté objets et traduit des préoccupations d'efficacité, de déterminisme, de sécurité et de génie logiciel. L'autre, en revanche, est interprété et propose un modèle d'exécution radicalement différent, non déterministe, une représentation uniforme des données et des programmes.

Prolog et la répartition

La répartition doit être traitée avec circonspection par Prolog. Ainsi, comme nous l'avons dit précédemment, les prédicats que nous avons ajoutés à Prolog ne réussissent qu'une fois au plus. On pourrait cependant s'interroger sur la signification d'un backtrack sur un envoi de message ou une migration. Notre réponse semble la plus sage dans un premier temps mais un problème tel que l'"annulation d'une communication" dans un environnement réparti peut constituer un sujet de réflexion intéressant.

L'échange de données Prolog-C++

La difficulté majeure de l'intégration concerne le partage de structures de données. En effet, les invocations et transmissions de données dans PUMA se font uniquement par le biais de chaînes de caractères. Que se soit par invocation synchrone ou par message, ce format pivot est la seule façon de communiquer entre C++ et Prolog, et nécessite une traduction à chaque envoi et à chaque réception.

La solution adoptée par SWI Prolog [WIE 94], qui fournit une interface C[5], consiste à mettre à la disposition du programmeur un ensemble de fonctions de conversion entre les

5. A l'inverse de notre intégration, la bibliothèque C fournie par SWI Prolog permet d'ajouter des prédicats définis en C qui prennent en compte le backtrack, mais l'invocation de Prolog depuis C ne permet pas de provoquer un échec pour déclencher un backtrack.

termes Prolog et les données C. Ainsi, l'invocation d'un but nécessite la construction du terme Prolog correspondant à l'aide de ces fonctions. Réciproquement, celles-ci réalisent le décodage des résultats retournés par Prolog. Cette solution est intéressante dans la mesure où l'on ne réalise qu'une traduction par échange puisqu'il n'y a pas de format pivot. Cela dit, la traduction entre chaîne de caractères et terme Prolog se fait de façon transparente dans PUMA et les manipulations de chaînes nécessaires au niveau du langage C pour créer ou interpréter des termes se font généralement assez aisément avec les fonctions standard de C.

6.2 Les classes dérivées

6.2.1 Arbre d'héritage

La classe `puma`, base de notre plate-forme, ne possède aucune activité. Sa seule utilisation directe ne peut donc consister qu'à créer des objets serveurs Prolog, invoqués de façon synchrone par des objets COOL quelconques. De surcroît, cette absence d'activité propre l'empêche de s'initialiser et de réagir aux invocations asynchrones de façon autonome.

Le rôle de cette classe est donc essentiellement de réaliser l'intégration d'un interpréteur Prolog dans COOL de la façon la plus transparente et générique possible en vue d'une utilisation par des classes opérationnelles dérivées.

6.2.2 La classe interp

La classe `interp` (cf. figure 24) hérite de `PUMA` mais définit une activité propre d'interpréteur interactif PUMA en exécutant la méthode `puma_interp()`. Elle permet de créer des interpréteurs Prolog classiques mais au dialecte étendu de capacités de communication. Ceux-ci peuvent être utilisés pour interagir dynamiquement avec un système d'objets COOL et, plus particulièrement, d'objets PUMA. Ils se révèlent donc précieux pour l'observation et l'aide à la mise au point d'objets PUMA plus évolués tels que les agents.

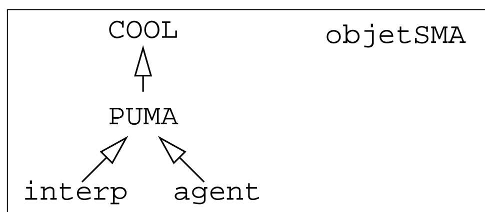


figure 24 : les classes PUMA standard

6.2.3 La classe agent

En créant une classe `agent`, nous tentons de définir une activité générique pour les objets PUMA qui soit adaptée à nos besoins. Elle consiste en une succession d'appels à un prédicat définissant l'étape élémentaire de l'activité de l'agent (cf. figure 25), en exclusion mutuelle. La principale motivation vient de l'impossibilité, pour des raisons pratiques, de coder l'activité de l'agent par la simple exécution d'un programme Prolog. En effet, en raison de l'exclusion mutuelle qui doit être maintenue sur le noyau Prolog, l'exécution d'un programme Prolog bloque toute invocation synchrone. Pour laisser le noyau disponible, il faut donc découper l'activité Prolog en étapes élémentaires, chacune s'exécutant de façon "atomique" en partant du "top level" de l'interpréteur Prolog.

```
void agent::main() // méthode définissant l'activité de l'objet
{
     puma_init();
     do {
          puma_P();
          if (pstate != P_END)
                 puma_call("activite.");
          puma_V();
     } while (pstate != P_END);
     puma_exit();
}
```

figure 25 : définition de l'activité "générique" d'un objet agent

De façon annexe, ce découpage en étapes élémentaires indépendantes et atomiques peut se révéler très utile en cas de "plantage" de la résolution d'un but Prolog, dû, par exemple, à la non-terminaison de la dérivation. Même en dehors de ce cas extrême, un débordement des structures internes de l'interpréteur Prolog (tas, piles globale/locale/auxiliaire etc) peut survenir en cours d'exécution. Certaines d'entre elles sont capables de s'agrandir dynamiquement mais d'autres sont dimensionnées de façon statique à la création. Un débordement provoque l'arrêt de l'exécution du programme Prolog et un retour au "top level". Avec notre principe de découpage, l'activité de notre agent se poursuit quoi qu'il arrive, même si une étape échoue. Bien entendu, le programmeur aura pour soucis :

- de dimensionner selon ses besoins les structures de données statiques de l'interpréteur et de restreindre les risques de débordement, comme pour tout programme Prolog ;

- de concevoir les agents et le système global de façon à minimiser l’interdépendance des étapes élémentaires, ceci afin de limiter les “sections critiques”[6] pouvant aboutir à un défaut de cohérence globale en cas d’interruption brutale ;
- de mettre en place un contrôle de l’exécution effective de ces “sections critiques”.

Cette solution simple paraît discutable, du fait de la granularité potentiellement grosse des étapes élémentaires, laissée aux soins du programmeur. Un mécanisme de type “temps partagé”, ou un découpage plus fin géré au niveau de la résolution, auraient assuré une transparence dans les invocations multiples. Mais la gestion du pseudo-parallélisme et des problèmes de cohérence aurait engendré une complexité particulièrement coûteuse, notamment en espace. Notre solution se justifie donc par son “économie”, et par le fait qu’elle nous paraît suffisante vis-à-vis de nos préoccupations. Notre soucis est de programmer une activité d’agent à tendance cognitive en Prolog mais de conserver un certain contrôle au niveau C++.

6.2.4 La classe objetSMA

La classe `objetSMA` est à rapprocher du principe de projection des entités d’un système d’information dans un univers homogène d’agents. Elle consiste à associer un agent PUMA à tout objet dérivant de COOL via un héritage. Cela signifie que tous les objets COOL d’un système vont être projetés dans une couche multi-agents intégratrice dans le but de favoriser leurs interactions.

Tout objet d’une classe `A` héritant de `objetSMA` se voit doté de deux méthodes et de deux attributs lui permettant d’exploiter l’agent automatiquement associé. Le constructeur de la classe `A` se doit d’appeler le constructeur de la classe `objetSMA` avec les arguments requis par la classe `agent` (i.e. le nom de l’agent et le fichier Prolog initial) pour que l’agent associé soit créé. De plus, ce constructeur initialise les deux attributs qui contiennent les adresses pour les invocations synchrones et asynchrones de l’agent.

L’une des deux méthodes sert à installer l’agent à la création ou à le réinstaller après migration. Ceci nécessite l’invocation de la méthode `puma_init()` de l’agent, mais aussi la mise en œuvre d’autres mécanismes visant à faire suivre l’objet potentiellement nomade par son agent, de la façon la plus transparente possible. Pour cela, l’agent doit prendre en compte

6. l’expression “section critique” ne doit pas être considérée ici dans son acception habituelle relative aux problèmes d’accès concurrents, mais plutôt selon un point de vue transactionnel.

un message “système” particulier qui le prévient de la migration de l’objet. Ce message est envoyé par la méthode d’installation qui reste bloquée en attente d’une réponse. L’agent peut alors prendre certaines précautions, puis migrer vers l’objet par simple appel au prédicat follow/0, qui a également pour effet de débloquer la méthode d’installation. Quant à l’autre méthode, elle sert à supprimer l’agent, notamment par appel à sa méthode `puma_exit()`.

Comparé à un objet `agent` seul, l’agent associé à un objet via les mécanismes de la classe `objetSMA` propose des fonctionnalités étendues. D’une part, on note que l’on obtient deux activités parallèles, typiquement l’une codée en C++ et l’autre en Prolog/PUMA, au lieu d’une seule. D’autre part, ses réflexes ont été étendus afin de simplifier et de rendre le plus transparent possible l’association avec un objet COOL quelconque. Ainsi, en plus du réflexe de création, l’agent peut définir des réflexes de post-migration et de pré-destruction.

6.3 Commentaires

6.3.1 PUMA et les agents

Qu’entendons-nous par le terme d’agent ? Cette question est importante pour la clarté de nos propos, car la notion d’agent recouvre des choses très variées. Les points clés de nos travaux relèvent de la répartition, de la mobilité, de l’activité autonome, du rôle d’intermédiaire coopérant, des capacités de communication et de raisonnement. Nous empruntons donc de très nombreuses notions à l’ensemble des approches présentées au Chapitre 4, et ce à divers niveaux :

- à un niveau conceptuel d’agent, représentant toute entité potentiellement cliente et/ou prestataire de service, disponible dans le système d’information (ressources matérielles ou logicielles, intervenants humains) ;
- à un niveau d’analyse, qui conduit à représenter chacune de ces entités par un agent afin de réaliser une couche homogène de coopération et de raisonnement (cf. Chapitre 5) ;
- au niveau de l’implémentation de cette couche sur un système réparti, avec des objets dont les caractéristiques particulières sont clairement dans la mouvance “agent”.

Vis-à-vis des agents tels que les envisagent les informaticiens et les industriels de la communication, les parallèles sont immédiats. Par exemple, la représentation d’une ressource par un agent, jouant un rôle de contrôle et d’intermédiaire, ressemble à la démarche SNMP. Du point de vue de l’outil PUMA, l’activité et la mobilité, ainsi que l’aspect langage interprété,

s'apparentent à certains aspects de Telescript. Enfin, le support de raisonnement logique rappelle que le besoin d'intelligence est une constante dans toutes les approches "agents" décrites.

En ce qui concerne l'IAD, PUMA apporte la possibilité d'implémenter des systèmes d'agents logiques communicants, grâce à des fonctionnalités de communication avancées (messages, groupes, migration, invocation synchrone locale) et au support Prolog. De plus, PUMA constitue une passerelle vers les systèmes répartis à objets, apportant ainsi de nouveaux outils mais aussi de nouvelles familles de problèmes à résoudre.

6.3.2 PUMA et les acteurs

Puisque nous présentons PUMA comme un outil d'IAD, il est important de le positionner par rapport aux "acteurs". A de nombreux égards, PUMA se prête bien à l'implémentation de leurs propriétés. En effet, les objets PUMA s'exécutent en parallèle et peuvent créer d'autres objets PUMA. De plus, ils communiquent par messages asynchrones, et disposent d'une boîte aux lettres dont l'adresse est librement communicable. Ceci permet de prendre en compte les notions d'accointances et de continuité. Par ailleurs, la programmation Prolog correspond bien à la notion de comportement redéfinissable.

En revanche, le haut degré de parallélisme du modèle d'exécution des acteurs n'est pas naturel dans PUMA, et ne peut être rendu de façon efficace. L'unique façon d'introduire du parallélisme entre les actions déclenchées par une communication reçue, consiste à créer autant d'objets PUMA. Le parallélisme peut donc être simulé par une couche logicielle supplémentaire (un module Prolog et une classe dérivée de PUMA) assurant ce mécanisme, mais la forte granularité des objets serait très pénalisante.

Pour illustrer nos propos, nous transposons deux exemples issus de [AGH 88] à l'aide de la classe `interp`.

6.3.3 Exemples

Calcul parallèle d'une somme

Cet exemple consiste à paralléliser le calcul de la somme de n nombres. Pour cela, on met en œuvre des objets élémentaires *additionneurs* (figure 26) qui traitent des requêtes formées d'une liste non vide de nombres et de l'adresse de l'objet devant collecter le résultat. En fait,

cette continuation est assurée par l'objet appelant. Une fois la requête traitée, l'objet additionneur disparaît.

Le traitement de la requête consiste :

- soit à calculer la somme directement, puis à envoyer le résultat, si la liste contient un ou deux éléments ;
- soit à créer deux objets additionneurs pour soumettre à chacun une moitié de la liste de nombres. L'objet attend les deux résultats[7], les additionne, puis envoie la somme à la continuation.

Calcul récursif parallèle de factoriel

Dans cet exemple, la continuation est effectuée par un objet consommateur. Il s'agit de paralléliser le calcul d'une factorielle par l'intermédiaire d'un objet permanent *factorial* dont le rôle est de réaliser la décomposition récursive $n! = n*(n-1)!$. Les appels récursifs se font par l'intermédiaire de messages que l'objet *factorial* s'envoie à lui-même, et dont la continuation est assurée par des objets temporaires *customer* (figure 27). A chaque étape de la décomposition, l'objet *factorial* crée un nouvel objet *customer* dont le rôle est d'attendre de résultat de l'étape suivante, pour le multiplier par n. n objets *customer* sont ainsi créés, en attente d'un résultat, jusqu'à ce que la récursivité se termine.

Lorsque n vaut 0, l'objet *factorial* envoie la valeur 1 au dernier objet *customer* créé comme résultat de 0!. Celui-ci peut alors effectuer la multiplication et transmettre le résultat à l'objet *customer* précédent. Successivement, tous les objets *customer* en attente se débloquent, calculent leur produit, transmettent le résultat, et disparaissent. Le consommateur terminal, objet quelconque ayant envoyé la requête initiale à l'objet *factorial*, reçoit le résultat final.

On note que l'objet *factorial* décompose le problème mais répartit entièrement la collecte des résultats, et ne centralise pas le résultat final. En outre, l'objet *factorial* est "réentrant", i.e. il peut prendre en charge plusieurs calculs différents en parallèle (même si une requête parvient en cours de décomposition d'une autre).

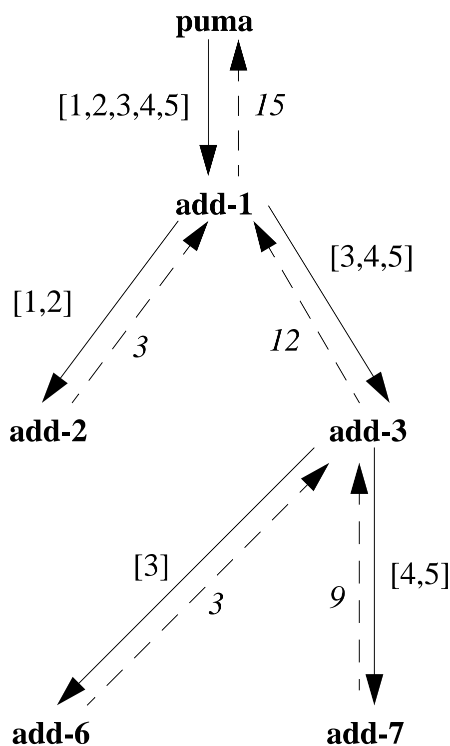
7. On voit donc que la continuation est assurée par l'objet appelant.

```
% fichier : addition
% comportement d'un additionneur

% puma
% réflexe de création
puma :-
     (repeat, receive([Id,L])),
     write('creation de '),
     myname(N),
     write(N),
     write(', calcul de '),
     write(L),
     nl,
     addition(L,R),
     send(Id,R),
     end.
puma :-
     myname(N),
     write(N),
     write(' : echec'),
     nl,
     end.

% addition(+L, ?S)
% S est la somme des éléments de L
addition([], 0) :- !.
addition([X], X) :- !.
addition([X,Y], Z) :-
     Z is X+Y,
     !.
addition(L,S) :-
     length(L, N),
     M is N // 2,
     sousliste(L,M,L1),
     append(L1,L2,L),
     myname(Agent-Id),
     Id1 is 2*Id,
     Id2 is 2*Id+1,
     create(interp, Agent-Id1, addition),
     create(interp, Agent-Id2, addition),
     (repeat, send(Agent-Id1, [Agent-Id, L1])),
     (repeat, send(Agent-Id2, [Agent-Id, L2])),
     (repeat, receive(S1)),
     (repeat, receive(S2)),
     S is S1 + S2.

% sousliste(+L, +N, ?R)
% R est la liste des N premiers éléments de L
sousliste(_,0,[]) :- !.
sousliste([X|L], N, [X|R]) :-
     N1 is N - 1,
     sousliste(L, N1, R).
```



$ **runp interp puma**
<puma> : **create(interp, add-1, addition).**
oui
<puma> : **send(add-1, [puma, [1,2,3,4,5]]).**
oui
<puma> : creation de add-1, calcul de [1,2,3,4,5]
creation add-2, calcul de [1,2]
destruction de add-2
creation de add-3, calcul de [3,4,5]
creation de add-6, calcul de [3]
destruction de add-6
creation de add-7, calcul de [4,5]
destruction de add-7
destruction de add-3
destruction de add-1
receive(Somme).
proposition : Somme = 15 :
accepter ou Refuser ?
oui
<puma> :

exemple d'exécution

figure 26 : exemple de calcul parallèle d'une somme par des objets PUMA

```
% objet factorial

puma :-
    assert(compteur(0)),
    repeat,
        receive(M),
        traite(M),
        fail.
% traite([+N, +Consommateur])
traite([0, U]) :-
    (repeat, send(U, 1)),
    !.
traite([N, U]) :-
    retract(compteur(C)),
    C1 is C + 1,
    assert(compteur(C1)),
    create(interp, cust(N, U, cust-C1), customer),
    myname(Self),
    N1 is N - 1,
    nl,
    send(Self, [N1, cust-C1]),
    !.
```

```
% objet customer

puma :-
    myname(cust(N, U, Name)),
    vanish,
    appear(Name),
    write(Name),
    write(' est cree.'),
    write(N),
    nl,
    (repeat, receive(K)),
    R is K * N,
    (repeat, send(U, R)),
    end.
```

figure 27 : calcul récursif parallèle de factoriel

6.4 Quelques détails pratiques de l'intégration

6.4.1 Encapsulation de C-Prolog

L'interpréteur C-Prolog a été modifié afin de pouvoir être compilé sous forme d'une bibliothèque contenant la fonction `pEvent()`, nécessaire et suffisante au contrôle de l'interpréteur. Cette fonction prend un type d'événement et des paramètres facultatifs en argument, et retourne un entier indiquant l'état de l'interpréteur Prolog, ainsi qu'une chaîne de caractères s'il y a lieu. Ainsi le prototype de cette fonction est :

```
int pEvent(pContext *PPtr, int func, char* in, char *out)
```

- `PPtr` pointe sur un enregistrement regroupant les “variables globales” de Prolog (cf. 6.4.3) ;
- `func` est le type de l'événement ;
- `in` est une chaîne de caractères contenant les arguments éventuels liés à l'événement : un but Prolog à résoudre, un nom de fichier “startup” (état Prolog initial), une clause Prolog à ajouter au programme, ou le résultat de l'invocation réussie d'un prédicat PUMA.

- `out` est une chaîne de caractères susceptible de retourner, suivant l'état de l'interpréteur, une proposition d'unification (couples variable/valeur), un message d'erreur Prolog, ou des arguments pour l'invocation d'un prédicat PUMA.

L'ensemble des états et des différents types d'événement est indiqué par le tableau 28. La

états		événements	
PREADY *PERROR*	interpréteur prêt (top level) *(+erreur Prolog)*	PCALL	résoudre un but
PYES	résolution réussie (succès)	PASSERT	ajouter une clause
POK	résolution possible	PCONT	retourner au top level
PFAIL	résolution impossible (échec)	PFAIL	déclencher un échec (backtrack)
PEND	session Prolog terminée	PSTART	initialiser Prolog (“startup”)
PFUNC	appel d'un prédicat PUMA	PFUNC	prédicat PUMA effectué, succès

tableau 28 : fonction pEvent(), états et événements

fonction `pEvent()` assure un contrôle “bas niveau” de Prolog et son utilisation directe est délicate car elle nécessite une bonne connaissance du fonctionnement interne. En effet, toutes les transitions ne sont pas possibles, et déclencher un événement incompatible avec l'état courant aboutit à des incohérences internes à l'interpréteur qui se “perd”. Il faut donc considérer l'interpréteur comme un automate, dont les principaux états et transitions sont données figure 29.

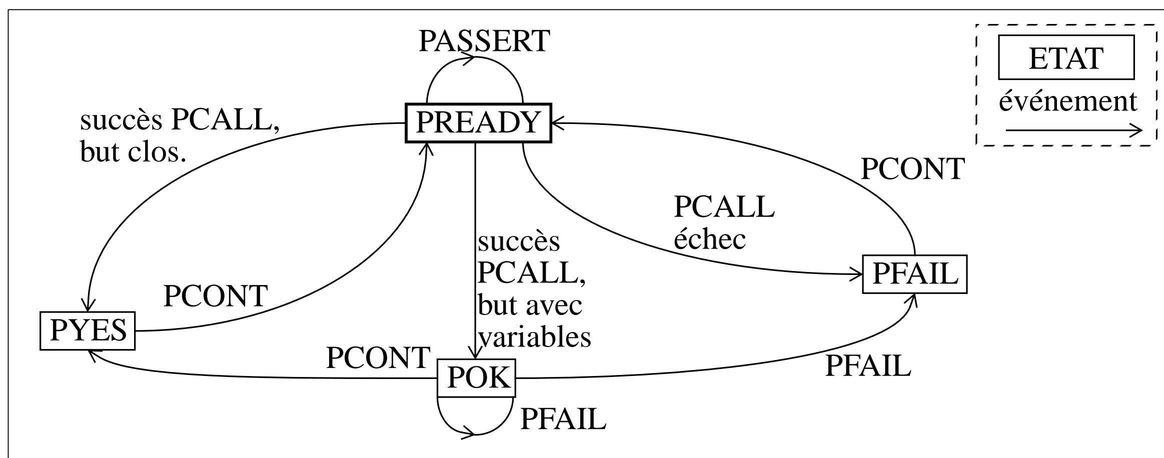


figure 29 : Comportement d'interpréteur Prolog standard de l'automate PUMA

6.4.2 Intégration à COOL

L'encapsulation de Prolog dans la fonction pEvent() nous permet de résoudre les deux problèmes posés par l'intégration de Prolog dans un objet COOL :

(1) pouvoir piloter l'interpréteur depuis C++ ;

(2) ajouter des prédicats mettant en œuvre des fonctions de COOL qui ne peuvent pas être directement implémentés dans Prolog mais doivent être programmés au niveau C++.

Pilotage du noyau Prolog

En ce qui concerne le point (1), nous avons rendu plus transparent le pilotage du noyau Prolog grâce aux méthodes de la classe `puma` qui se chargent de gérer les états et les événements via `pEvent()`. Seul un sous-ensemble minimal d'états fondamentaux est visible par le programmeur, pour la gestion de ses invocations Prolog.

Les méthodes fournies agissent elles-mêmes en automate pour répondre aux invocations du programmeur qui les utilise, ainsi qu'aux sollicitations provenant de pEvent(). En effet, en plus de protéger l'interpréteur contre des événements incohérents avec l'état courant, elles permettent de masquer les appels aux prédicats PUMA.

Invocation d'un prédicat PUMA

Le point (2) est traité dans le cadre de la famille des états “PFUNC”. Dans C-Prolog, chaque prédicat “système” est représenté de façon interne par un identificateur. Nous y avons ajouté notre famille de prédicats système PUMA, repérés par des identificateurs réservés. Lorsque l'interpréteur Prolog rencontre l'un de ces prédicats, la fonction `pEvent()` retourne dans un état PFUNC spécifique, dans lequel est encodé l'identificateur associé. Le niveau C++ n'a donc plus qu'à exécuter la partie de programme correspondante, sachant que les arguments de ce prédicat sont transmis via une chaîne de caractères passée en argument à l'appel de `pEvent()`. Suivant l'événement déclenché par l'appel `pEvent()` suivant, on peut signifier soit un échec, soit un succès avec une éventuelle valeur pour l'unification (voir exemple figure 30).

6.4.3 Les difficultés rencontrées

Outre la transformation de la boucle principale de l'interpréteur, nécessaire à la création de la fonction `pEvent()` ainsi qu'à l'introduction de prédicats “système” externes (prédicats PUMA), il a fallu regrouper l'ensemble des variables globales dans un enregistrement afin de

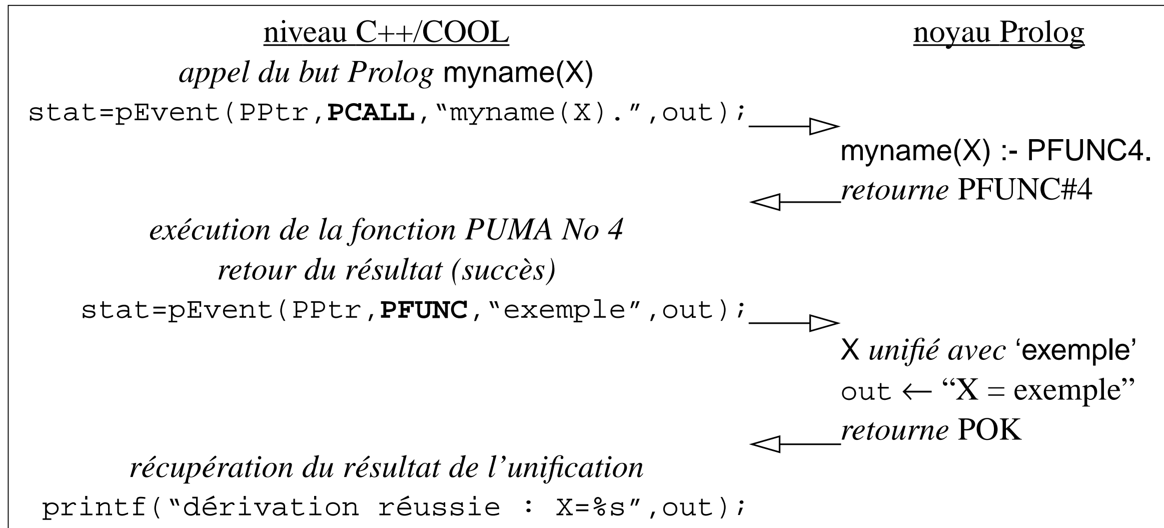


figure 30 : myname/1 - détails de l'invocation d'un prédicat PUMA

les déporter de C-Prolog vers les objets C++/COOL (cf. paramètre `PPtr` de `pEvent()`). En effet, de telles variables auraient été considérées par COOL comme des pseudo-"variables de classe", partagées par les objets d'une même classe au sein d'un même contexte. Ceci aurait interdit la coexistence de plusieurs objets PUMA dans un même contexte.

En ce qui concerne la migration d'un objet PUMA, elle présente la particularité de conserver l'activité Prolog en cours. Outre la migration standard de l'objet COOL correspondant, cela nécessite la sauvegarde de l'état Prolog dans un fichier, puis sa restauration à distance sur le site de réinstallation. Mais les principales difficultés posées par la migration concernent les communications.

D'une part, l'objet migrant doit prendre des précautions vis-à-vis des communications synchrones éventuellement engagées, puisqu'elles seront rompues. D'autre part, il doit prendre soin de ne pas perdre de messages. Ceci implique non seulement la sauvegarde des messages en attente dans sa boîte aux lettres[8], mais également le retrait du service de nommage afin d'éviter que des messages ne lui soient envoyés au cours de sa migration. Comme il faut tenir compte d'une certaine inertie du système (e.g. délai d'acheminement d'un message), une temporisation est nécessaire entre le retrait du service de nommage et la sauvegarde de la boîte aux lettres. Le réglage empirique de ce délai, ainsi que la "disparition" temporaire de l'objet, sont des solutions certes opérationnelles, mais évidemment peu satisfaisantes.

8. Ces messages sont lus, stockés au sein de l'objet, puis retransférés dans la boîte aux lettres après migration. L'option de migration préservant la file de messages, prévue dans Chorus et COOL, n'est pas encore disponible.

6.4.4 Le coût des objets PUMA

La bibliothèque Prolog contenant la fonction `pEvent()` n'a pas été directement intégrée au code de la classe `puma`. En effet, cela aurait engendré une surcharge des objets, notamment coûteuse en cas de migration. Par conséquent, nous avons intégré cette bibliothèque aux contextes d'exécution COOL. Il est en résulte un faible encombrement pour les classes `interp` et `agent` (cf. 6.2.2 et 6.2.3), réduit à 17K.

Le contexte d'adressage, qui contient aussi la bibliothèque COOL, totalise 120K. Les allocations dynamiques proviennent essentiellement de l'interpréteur Prolog. Lorsque les programmes Prolog sont chargés, la place mémoire occupée avoisine les 100K. Ce chiffre permet d'estimer le nombre d'agents que l'on peut créer sur un site, ainsi que le coût de la migration par rapport au simple envoi de message.

6.5 Conclusion

Basée sur l'intégration d'un interpréteur Prolog dans les objets du système réparti COOL, la plate-forme PUMA propose une approche mixte : dialecte Prolog étendu de prédicats de communication, et objets C++ disposant de méthodes de résolution logique. Il en résulte un outil original présentant plusieurs facettes :

- plate-forme d'implémentation de systèmes d'agents logiques communicants et répartis ([DIL 94]) ;
- extension de l'approche orientée objets offrant une passerelle entre l'IAD et les systèmes répartis à objets ([DIL 95a]) ;
- outil de prototypage et interpréteur de commande ("shell") pour les systèmes répartis à objets.

Pour tous ces points, PUMA montre la faisabilité et l'intérêt de la démarche, par un premier niveau d'outils, mais ne constitue qu'une première étape. En ce qui concerne l'IAD, le type de logique proposé (premier ordre) peut s'avérer insuffisant pour prendre en compte "la nature des connaissances, dans un système hétérogène en activité où des agents autonomes, à capacités limitées, ayant des points de vue partiels et nécessairement subjectifs sur leur environnement, communiquent et collaborent" [LEF 95]. Du point de vue des systèmes répartis, l'évolution de COOL doit être prise en compte. Aussi les travaux se poursuivent-ils sur COOL v2 : [HYM 95] présente une intégration similaire réalisée avec LIFE[9], et [MAL 95] choisit une approche complémentaire sous forme d'un serveur Oz[10] externe aux objets.

Pour notre part, c'est l'aspect "passerelle IAD/systèmes répartis à objets" qui nous préoccupe particulièrement. Comment une approche de type IAD peut-elle résoudre des problèmes liés aux systèmes répartis à objets ? Réciproquement, dans quelle mesure ces mêmes systèmes répartis à objets permettent-ils d'implémenter des systèmes "multi-agents" ? Nous abordons cette double problématique dans un contexte de bureautique communicante.

---

9. "Logic, Inheritance, Functions and Equations" : langage interprété dérivé de Prolog, intégrant des principes de programmation fonctionnelle, de concurrence, ainsi que de contraintes liées à une notion de type (cf. logique *sortée*) et d'héritage.
10. Oz combine programmation fonctionnelle, orientation objet, logique d'ordre supérieure (unification sur les prédicats), contraintes, typage dynamique, avec un certain degré de parallélisme.

Chapitre 7

Exemples de mise en œuvre

Dans ce chapitre, nous présentons la mise en œuvre de PUMA et de l'approche multi-agents des systèmes de bureautique communicante, à travers deux maquettes. La première est minimale du point de vue de la constitution des agents, mais permet de montrer certaines caractéristiques propres à l'utilisation de PUMA, et de percevoir plus distinctement l'intérêt de la démarche multi-agents dans le cadre du *Workflow*. La seconde maquette exploite de façon plus aboutie le principe de représentation des ressources du système par des agents coopérants, capables de servir de support à tout type d'application de bureautique communicante multi-agents. Elle introduit également la notion de structure et de protocole de coopération, qui fait l'objet du chapitre suivant.

7.1 Premier pas : Workflow multi-agents

7.1.1 Présentation

Cet exemple préliminaire constitue un prototype volontairement minimal de la vision multi-agents des systèmes de bureautique communicante. Inspiré de CIDRE (voir Chapitre 2), le système met en présence deux catégories d'agents :

- l'agent *impresario*, représentant un utilisateur, chargé de répondre aux requêtes ;

- l'agent *document*, qui invoque les agents impresarios pour que certaines actions soient réalisées par les utilisateurs.

L'enchaînement des actions à exécuter sur un document est décrit par un ICN (voir Chapitre 2 et Annexe B). Elles consistent typiquement à remplir un champ du document, mais il peut s'agir d'autres traitements, comme par exemple une impression du document ou un visa à accorder. Deux versions ont été réalisées, l'une mettant en jeu une exécution asynchrone par envoi de messages (requêtes/réponses), l'autre procédant par pseudo-parallélisme et migration de l'agent document (voir [DIL 92]). Nous présentons plus particulièrement la première, qui présente l'avantage de conserver un réel parallélisme d'exécution.

7.1.2 Les agents

Principe

A plusieurs titres, cet exemple est restrictif par rapport à notre approche multi-agents. Notamment, le comportement de nos agents est dissymétrique puisque les agents documents sont des clients purs, et les agents impresarios des serveurs purs.

Ainsi, l'agent impresario exécute une boucle infinie consistant en une tentative de lecture et de traitement d'un message, alternée avec une tâche propre. Celle-ci est réduite à sa plus simple expression mais constitue un point d'ouverture pour étendre l'activité de l'agent. Le traitement typique d'un message consiste à demander une valeur à l'utilisateur (son nom, une date…) et à la retourner au document. De toute façon, toute action demandée implique qu'un message d'avis d'exécution soit renvoyé par l'impresario.

Quant à l'agent document, il exécute l'ICN suivant un principe d'envoi de requêtes asynchrones. L'activité est analogue à celle de l'impresario, mais les messages traités correspondent à des réponses de requêtes. L'état du document, représenté sous forme d'un historique construit par l'agent, est défini par :

- une liste d'actions déclenchées,
- une liste d'actions terminées,
- la valeur de certains champs.

A chaque message reçu, le document recherche et déclenche les nouvelles actions rendues possibles par l'accomplissement de l'action concernée. Puisque l'ICN est un treillis, la fin du

parcours est simplement repérée lorsqu'une action sans nœud successeur se termine. L'activité de l'agent s'arrête alors.

Implémentation

Ce système est antérieur à la définition de la classe `agent`. Il met en œuvre la classe `interp` et l'activité de l'agent est démarrée par l'intermédiaire du réflexe de création[1]. La création d'un agent consiste à invoquer une commande particulière ("runp", voir Annexe A) en fournissant en argument :

- la classe C++/COOL de l'objet à créer ;
- le nom symbolique de cet objet (cf. service de nommage COOL), sachant que le nom des objets doit correspondre au nom des intervenants spécifiés dans l'ICN ;
- le nom d'un fichier Prolog à charger.

Cet ultime paramètre détermine le type d'agent créé, suivant que l'on charge le fichier *impresario* ou *document* (voir Annexe B). Le schéma de l'activité, semblable dans les deux cas, est spécialisé par les types de message reconnus et leur traitement associé. Cela revient à définir de façon spécifique une procédure traite/1, dont l'argument décrit une sorte de format devant s'unifier avec le message reçu. L'agent document est plus complexe que l'agent impresario, car il intégre la définition déclarative de l'ICN et l'ensemble des procédures nécessaires à son exécution. En particulier, l'agent document procède à une analyse préliminaire de l'ICN afin de repérer les possibilités de parcours bloquants. Ceci permet de les éviter en cours d'exécution, ou de les détecter de façon anticipée.

7.1.3 Bilan

Ce système est très clairement simplifié et incomplet, mais son implémentation laisse apparaître un certain nombre de choses :

- Un schéma simple et viable d'activité, consistant en une tentative de lecture et de traitement de message alternée avec une activité propre élémentaire. Toutefois, il est souhaitable de créer une activité parallèle dans le cas d'opérations "longues".
- Un typage double des agents, qui s'opère à partir de la conjonction d'une classe C++/PUMA, et d'un fichier Prolog. L'interface Prolog, définie par l'ensemble des types de

1. L'activité est donc entièrement exécutée au niveau Prolog, sans aucun passage au top-level par le niveau C++, ce qui interdit les communications synchrones.

messages reconnus, est capable d'évoluer (disponibilité de nouveaux services), ce qui ouvre la voie à un "typage" dynamique.

- Le rôle clé de l'agent représentant une ressource. D'une part, il apporte une uniformité de l'accès aux services, un agent similaire aux impresarios pouvant représenter une ressource quelconque (e.g. une imprimante pour l'action "imprimer"). D'autre part, il joue un rôle d'assistant "intelligent" envers sa ressource, pour peu qu'il ait un certain niveau de connaissance sur celle-ci. Ainsi, l'impresario pourrait proposer des réponses automatiques lorsqu'il s'agit de donner le nom, le service ou la date.
- Le besoin de structures de coopération intermédiaires capables de gérer l'indirection entre le nom d'un agent particulier et un rôle générique.
- Le support adapté de Prolog pour un maquettage rapide (programmation déclarative, langage interprété, typage des messages, représentation de l'ICN) et pour implémenter des capacités de raisonnement (cf. analyse de l'ICN).

7.2 Principe d'un système de réservation de salles multi-agents

7.2.1 Présentation générale

Il s'agit de mettre en œuvre la modélisation multi-agents à l'aide des outils présentés, dans un cas concret et simple mais de façon beaucoup plus complète qu'en 7.1. A travers l'exemple d'un système de réservation de salles, nous souhaitons mettre en évidence une base opérationnelle multi-agents permettant l'intégration de toute sorte d'application de bureautique communicante. Les agents mis en jeu dans notre système ne sont pas dédiés exclusivement à la gestion de réservation de salles. Moyennant une extension de leurs services et caractéristiques, les agents peuvent être le support d'une rédaction coopérative, d'un agenda réparti, d'une circulation de document, d'une messagerie intelligente…

Les ressources impliquées dans notre système sont les salles et les utilisateurs. Chacune de ces ressources est représentée de façon exclusive par un agent. Celui-ci joue un rôle d'interface complète entre la ressource et le reste du système (i.e. les autres ressources). Ainsi, lorsqu'un utilisateur désire réserver une salle, ou annuler une réservation, il a accès au service correspondant via son *agent utilisateur* qui se charge de contacter les *agents salle*. L'agent propose également un ensemble de services locaux, disponibles via une interface utilisateur d'administration.

Pour pouvoir représenter une ressource, les agents ont un rôle de mini-base de données locale. En effet, ils maintiennent un ensemble d'informations sur la ressource qu'ils représentent. La notion de caractéristique structurelle ou conjoncturelle est prise en compte par les agents, afin d'améliorer l'efficacité de la recherche d'un serveur[2]. Ainsi, les dimensions d'une salle sont typiquement invariables, alors que le planning des réservations évolue sans cesse. De même, la fonction d'un utilisateur possède un caractère statique que ne revêt pas son agenda. Ces propriétés interviennent au niveau des protocoles de coopération.

7.2.2 Les agents et les services

Les agents de ressource

Les agents *salle* et les agents *utilisateur* représentent les ressources du système. Ils possèdent donc un caractère permanent, lié à la présence de ces ressources. Les agents ont une représentation interne de leur ressource à travers des caractéristiques. L'agent salle connaît l'emplacement, la surface, et l'équipement (mobilier, dispositifs de projection…) de la salle. Il tient également à jour le planning des réservations. L'agent utilisateur connaît l'état civil, le bureau, le service ainsi que la fonction de l'utilisateur, et maintient un mémento des réservations qu'il a effectuées.

Chaque agent de ressource possède une *interface de contrôle*, qui permet à un utilisateur de l'administrer et de l'invoquer de façon directe et locale. Cette interface propose trois opérations :

- la destruction de l'agent, signifiant le retrait définitif de la ressource ;
- le déplacement physique de l'agent, qui permet de le transférer sur une autre machine ;
- l'invocation de services disponibles dans le système multi-agents.

L'invocation locale de l'agent, i.e. par l'intermédiaire de l'interface de contrôle, permet d'accéder à un ensemble de services représentant l'*interface privée* de l'agent (figure 31). Celle-ci propose un certain nombre de services que l'agent sait effectuer par lui-même, mais permet aussi d'accéder à des services fournis par d'autres agents du système, via leur *interface publique*. Dans ce cas, l'interface privée met en œuvre des protocoles associés à des structures de coopération particulières pour rechercher le service parmi les autres agents.

2. voir protocole du groupe d'artisans, Chapitre 8.

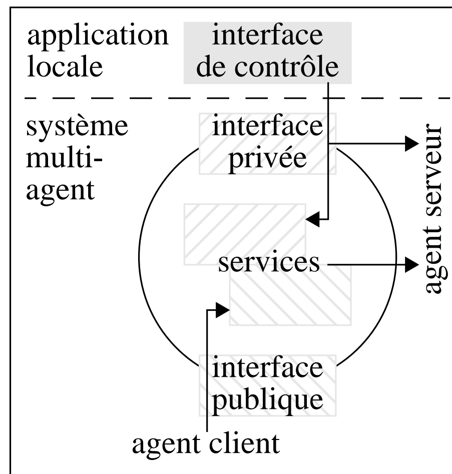


figure 31 : structure d'un agent de ressource

Les agents auxiliaires

Le système met également en jeu des *agents auxiliaires*, qui sont créés pour accomplir des tâches spécifiques liées à certains services, et disparaissent ensuite. Par exemple, le service de mémento[3], service interne offert par tout agent utilisateur, nécessite la création d'un agent auxiliaire dit *agent de service*. En effet, l'exécution de ce service, qui est interactif avec l'utilisateur, ne doit pas perturber le fonctionnement de l'agent utilisateur. On peut ainsi avoir plusieurs services qui s'exécutent en parallèle, voire plusieurs instances indépendantes d'un même service, chacune étant pris en charge par un agent. De même, la négociation du service de réservation met en jeu un agent auxiliaire, dit *agent négociateur*, créé par un agent salle et qui migre auprès du client. Les agents auxiliaires ne représentent pas directement de ressource, possèdent un rôle temporaire et autonome, et ne nécessitent donc pas d'interface de contrôle.

Les services

Les services proposés par un agent de ressource sont privés ou publics, suivant qu'ils sont accessibles par les interfaces correspondantes. Certains services s'exécutent de façon entièrement interne à l'agent, alors que d'autres nécessitent l'invocation d'un service intermédiaire sur un autre agent. Par exemple, tout agent de ressource accomplit de façon strictement interne des services privés de consultation et de modification de ses caractéristiques (figure 32 : "consulter-base", "modifier-base") ; par contre, l'agent utilisateur propose un service privé d'annulation de réservation de salle qui nécessite une invocation du service ad hoc sur l'interface publique de l'agent salle concerné ("annuler-salle").

3. Ce mémento permet à l'utilisateur de visualiser et rechercher certaines informations concernant les services qu'il a invoqués au sein du système multi-agents, comme par exemple les réservations de salle (caractéristiques de la salle réservée, heure, date).

L'agent de ressource permet également d'invoquer des services purement externes. C'est le cas du service de réservation de salle lorsqu'il est invoqué sur l'interface privée d'un agent utilisateur. Non seulement l'agent utilisateur ne connaît pas les agents salle disponibles, mais il est de surcroît incapable de formuler une requête complète de réservation. Il fait donc appel à des protocoles de recherche de serveur, associé à une structure de coopération particulière, et, par négociation avec l'utilisateur, obtient la salle la plus adaptée aux besoins (cf. Chapitre 8).

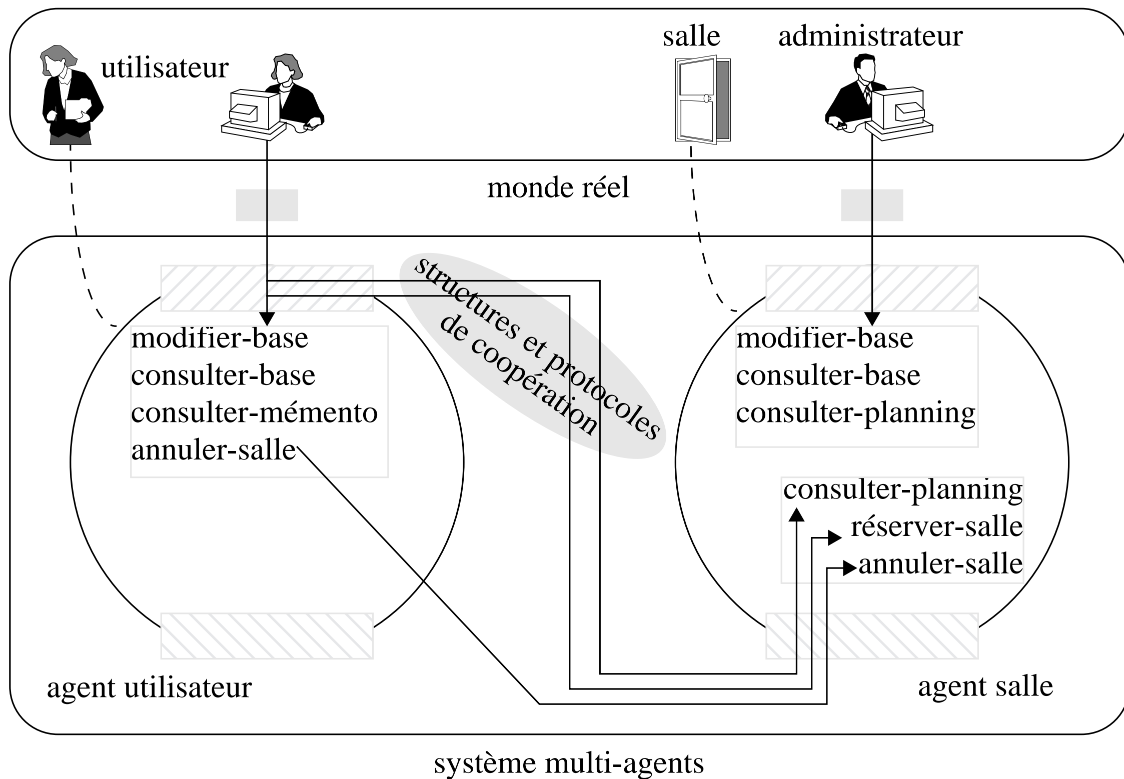


figure 32 : agents et services du système de réservation de salle

7.2.3 Négociation et résolution de contraintes

Principe

Pour décrire sans ambiguïté et négocier les services offerts ou recherchés, les agents partagent une sémantique liée à un certain nombre de symboles désignant des services (e.g imprimer, réserver-salle), ou des caractéristiques (e.g. résolution, surface). La sémantique attachée à un symbole de caractéristique dépend du domaine du service demandé, mais est unique pour ce domaine. Conformément à notre approche présentée au Chapitre 5, un agent client ne connaît pas, a priori, toutes les caractéristiques liées au service qu'il cherche. Cette éventualité est prise en compte par l'intermédiaire d'un agent auxiliaire, qui se charge de négocier avec le client pour préciser le “cahier des charges”. Ce mécanisme se substitue au

principe de contrainte qualitative et de caractéristique floue, qui n'a pas été mis en œuvre ici, mais il relève des mêmes préoccupations.

Les agents disposent également d'un langage d'expression de contraintes sur les caractéristiques. Sous-ensemble du langage décrit au Chapitre 5, il permet d'exprimer l'appartenance ou la non-appartenance d'une caractéristique à un intervalle ou à une énumération de valeurs, ainsi que les notions de maximum et de minimum. Le moteur de résolution de contraintes réalise une combinaison des contraintes du client et du serveur (vérification de la compatibilité et simplification), puis applique les règles de valuation par défaut du serveur, s'il y a lieu. La satisfaction est alors évaluée en fonction de la valeur des caractéristiques résolues. Pour plus de détails sur le moteur, on se reportera à l'Annexe D, qui apporte notamment des précisions sur les procédures génériques et la façon de les compléter par des clauses spécifiques.

Ce moteur est assez particulier (et simplifié), puisqu'il n'inclut pas d'algorithme de recherche de solution optimale, mais autorise, en revanche, tout type d'expression pour calculer la satisfaction[4]. Ainsi, dans notre exemple, les valeurs des caractéristiques des salles (surface, équipement…) sont fixées à l'avance par chaque agent salle, et il s'agit uniquement d'en vérifier la compatibilité avec les contraintes exigées par l'agent client. Pour choisir le meilleur serveur (i.e. la meilleure salle), il faut évaluer toutes les offres potentielles en calculant le résultat de la fonction de satisfaction dont les paramètres sont déjà valués. On pourrait songer à étendre ce fonctionnement en prenant en compte les situations où certaines caractéristiques du service offert peuvent avoir plusieurs valeurs différentes (par exemple, un agent imprimante pourrait offrir plusieurs qualités d'impression). Il faudrait alors engendrer toutes les solutions et évaluer les satisfactions correspondantes[5], dans l'hypothèse où le nombre de solutions possibles reste faible.

4. Dans la plupart des systèmes de résolution de contraintes, l'algorithme de recherche de solution optimale n'accepte que des expressions linéaires.
5. Toutefois, il faut noter qu'une telle extension impose de reconsidérer certains aspects du langage de contraintes initial. En effet, l'introduction du principe d'optimisation de fonction de satisfaction peut modifier légérement la sémantique attachée aux contraintes “minimum” et “maximum”, ainsi qu'aux règles de valuation par défaut.

Exemple

L'exemple qui suit, issu du système de réservation de salle, n'utilise que les contraintes d'appartenance à une liste (@) et de borne inférieure (>=). Pour une description complète du langage de contraintes, on se référera à l'Annexe D.

(1) contraintes exigées par le client :

[[client, @[bruno]], % le client indique son nom

[chaises, >=10], % il faut au moins 10 chaises

[tables, >= 6]] % il faut au moins 6 tables

(2) contraintes imposées par le serveur :

[[serveur, @[aviso]], % le serveur indique son nom

[chaises, @[20]], % la salle dispose de 20 chaises

[tables, @[12]]] % la salle dispose de 12 tables

(3) les contraintes étant compatibles, leur fusion donne le résultat suivant :

[[client, @[bruno]],

[serveur, @[aviso]],

[chaises, @[20]],

[tables, @[12]]]

(4) la valuation peut être effectuée par les règles de résolution génériques, puisque chaque contrainte définit une appartenance à un singleton :

[[client, bruno],

[serveur, aviso],

[chaises, 20],

[tables, 12]]

(5) connaissant les contraintes du client, l'agent négociateur établit un critère de satisfaction[6] qui consiste à minimiser l'équipement non utilisé :

[satisfaction(X), X is 1 / (1 + (&chaises-10)/&chaises + (&tables-6)/&tables)]

(6) A partir de la valuation des caractéristiques du service, la satisfaction peut être évaluée à 0,5. Elle serait maximale, i.e. égale à 1, si le nombre de chaises et de tables utilisées correspondait exactement aux besoins. Si un autre agent salle propose un équipement moindre mais suffisant, la satisfaction correspondante sera plus grande.

---

6. Le préfixe “&” signale que le symbole suivant est un nom de caractéristique qu'il faut substituer par sa valeur au moment du calcul.

7.2.4 Structure et protocoles de coopération

La coopération entre les agents est basée sur des messages de requête de service. Une requête complète comprend trois éléments : un nom de serveur, l'identificateur du service, et une liste de contraintes sur les caractéristiques de ce service. Une telle requête, envoyée explicitement à un agent déterminé, donne lieu à un message de refus ou à un rapport exprimant les conditions exactes d'acceptation du service. Ainsi, une requête de réservation de salle peut échouer, si les contraintes du client ne sont pas satisfaisables, ou aboutir à un rapport spécifiant les valeurs exactes des caractéristiques : surface de la salle, équipement précis…

Lorsqu'un agent recherche un service mais qu'il ne connaît pas de serveur ad hoc, ou qu'il souhaite obtenir le "meilleur service" parmi l'offre de tous les serveurs, il doit avoir recours à des protocoles et des structures particuliers. En effet, le nombre potentiellement conséquent d'agents interdit le principe d'une diffusion globale. De plus, la dynamique du système (apparition, disparition, modification d'agents) empêche les agents de tenir à jour des accointances exhaustives et fiables.

Notre système fait appel à une structure basée sur une notion de groupe de serveurs. Détaillée au chapitre suivant, elle consiste à organiser les agents salle en un groupe d'"artisans solidaires". Chaque salle connaît de façon exhaustive et cohérente les autres salles et leurs caractéristiques structurelles (équipement, surface…). Le planning des réservations, considéré comme insuffisamment stable, n'est pas diffusé (caractéristique conjoncturelle). Le groupe en tant qu'entité est enregistré sous un nom pré-défini, qui doit être connu par l'agent utilisateur. Le protocole associé à cette structure permet de contacter une salle aléatoire, qui peut ensuite réorienter la requête sur la salle dont l'équipement est le plus adapté. En cas de refus (incompatibilité avec le planning), un autre agent salle structurellement satisfaisant est contacté, et ainsi de suite.

7.3 Implémentation du système de réservation de salle

7.3.1 Conception des entités

Principe

Comme nous l'avons souligné dans l'exemple précédent, le type d'un objet PUMA résulte du croisement entre une des classes C++/COOL disponibles dans l'environnement PUMA, et une base Prolog initiale. La classe fournit une structure d'accueil "système" qui détermine le

potentiel d'une entité, dont le comportement, l'activité, les connaissances et les interactions sont programmés dans la base Prolog. La complexité variable des fonctions des différentes entités du système nous a conduit à exploiter toutes les classes PUMA, de la plus simple — `interp` — à la plus élaborée — `objetSMA` —.

Parmi les bases Prolog, on distingue des "modules"[7], qui définissent des capacités standard utilisables au sein de plusieurs types d'entité, selon les besoins. Ainsi, un module particulier permet de conférer des capacités individuelles d'agent PUMA suivant plusieurs plans :

- procédures utilitaires telles que la gestion d'identificateurs uniques d'objet ou la manipulation de fenêtres de dialogue.
- instanciation de l'étape élémentaire de l'activité de l'agent par définition de la procédure activite/0. Comme le montre la figure 33, celle-ci fait appel à une procédure de traitement d'un message (traite_message/1) et à une tâche élémentaire propre à l'agent (mon_activite/0).

```
% activite/0 : étape élémentaire de l'activité de l'agent (cf. classe d'objet 'agent')
activite :-
    lire_message, % traiter un éventuel message de la boîte aux lettres
    mon_activite. % tâche personnelle

% lire_message/0 : tentative de lecture et de traitement d'un message, lié à la définition
% d'une procédure traite_message/1. Réussit toujours.
lire_message :-
    receive(Message), % on tente de lire un message
    traite_message(Message). % appel à la procédure de traitement des messages
lire_message.

% traite_message/1 : procédure de traitement d'un message
traite_message(puma(ringup(Agent))) :- % prise en compte d'une connexion synchrone
    !,
    accept, % toute demande est toujours acceptée
    recordz(puma, connexion(Agent), _). % on mémorise l'existence de la connexion
traite_message(puma(hangup(Agent))) :- % prise en compte d'une déconnexion
    !,
    recorded(puma, connexion(Agent), Reference),
    erase(Reference).
```

figure 33 : schéma d'étape élémentaire de l'activité d'un agent

- définition d'un comportement de base par la prise en compte de certains messages (messages "système" puma/1, objetSMA/1 — voir Annexe A). Par exemple, toute requête

7. Cette notion de "module" est spécifique à nos propos (notre implémentation ne dispose pas de "modules" au sens Prolog), mais le choix de ce mot s'inspire néanmoins de son acception courante.

de connexion synchrone est acceptée et enregistrée. Pour des raisons de commodité, en particulier pour le “débogage”, le traitement des messages de “type” (i.e. s’unifiant avec) call(But) déclenche la résolution de But.

- gestion des connaissances sur les autres agents. Ceci établit une manière standard de représenter les caractéristiques des autres agents et de soi-même, ainsi que les procédures qui définissent l’interface d’accès à ces données. C’est ainsi que les agents peuvent récupérer des données au sein d’un autre agent par invocation synchrone.
- définition de services que tout agent est susceptible d’offrir. Ici, il s’agit des services privés de consultation et de modification des caractéristiques proposés par les agents de ressource (i.e. salle et utilisateur).

Un module lié aux capacités de négociation regroupe la définition du langage d’expression de contraintes, le moteur de résolution et les règles génériques correspondantes. Enfin, deux autres modules concernent les capacités de coopération. L’un contient les procédures de gestion interne de la structure de groupe, et l’autre permet son invocation pour la recherche d’un agent serveur.

Les entités sont entièrement définies par la base initiale qui charge les modules nécessaires et spécialise le comportement : extension de la procédure traite_message/1, ajout de services privés ou publics spécifiques, définition de la tâche personnelle mon_activite/0 et des réflexes[8].

Les objets utilitaires

Les entités les plus simples sont réalisées à partir de la classe `interp` et leur fichier Prolog initial n’inclut aucun “module agent”. Leur activité correspond à l’exécution d’un programme Prolog, lancé automatiquement par définition du réflexe de création puma/0. L’unique représentant de cette catégorie d’entité est l’*objet terminal*. Il consiste en une fenêtre de dialogue, capable d’afficher du texte et d’interroger l’utilisateur. Son activité se limite à une boucle infinie d’attente et de traitement de messages. Ceux-ci contiennent des requêtes d’affichage et de saisie de texte. On peut évidemment envisager d’y substituer tout type d’interface graphique rendant les mêmes fonctionnalités. De plus, comme ces objets sont construits à partir d’une définition locale, cela offre la possibilité de personnaliser l’interface.

8. Les réflexes sont représentés par les procédures puma/0 pour toutes les classes, et objetSMA(init), objetSMA(reinit), objetSMA(exit) pour les classes dérivant de `objetSMA` (cf. Annexe A).

Le développement d'une interface "intelligente" pourrait conduire à transformer cet objet utilitaire en agent.

Les agents de service

Les agents de service permettent d'exécuter un service pour le compte d'un agent maître. Ils sont constitués d'un objet de classe `agent` et des modules Prolog définissant les capacités individuelles d'agent. Ils ne chargent pas les modules de coopération, inutiles pour leur rôle d'exécutant. Ils permettent de lancer une activité parallèle à un agent de ressource, et d'offrir une interface utilisateur (via un objet terminal) spécifique au service. Ainsi, les services de consultation de mémento, de modification des caractéristiques, et d'annulation de réservation sont pris en charge par des agents créés à la demande, en nombre quelconque. Ces agents de service accèdent aux données de l'agent par invocation synchrone (phone/1), ce qui permet une grande efficacité dans la programmation ainsi que dans l'exécution. De plus, cela évite toute redondance et incohérence.

Les agents négociateurs

Les agents négociateurs ont un rôle comparable aux agents de service, mais sont plus élaborés et autonomes. Egalement conçus à partir de la classe `agent`, ils intégrent en plus les modules Prolog de négociation et de coopération afin d'assister un client dans la recherche d'un serveur. Ces agents ont un caractère nomade, puisqu'ils migrent auprès du client afin de rendre efficaces les interactions et d'accéder de façon directe et transparente à certaines caractéristiques par invocation synchrone (phone/1).

Les agents de ressource

Les agents de ressource constituent un exemple de mise en œuvre de la classe `objetSMA` (cf. Chapitre 6 et Annexe A). En effet, ces agents sont construits à partir de la classe `interface` (voir Annexe C), qui hérite de `COOL` et de `objetSMA`. Rappelons que tout objet de la classe `objetSMA` crée automatiquement un objet de la classe agent dans son constructeur, et possède les adresses d'invocation synchrone et asynchrone pour invoquer cet agent. De plus, l'agent suit automatiquement son créateur en cas de migration et possède des réflexes de création, migration et destruction.

Dans le cas de la classe `interface`, l'héritage de la classe `objetSMA` permet de définir deux entités et deux activités parallèles. L'objet interface est le maître, qui gère les interactions

avec l'administrateur de la ressource concernée, et l'agent représente de façon permanente cette ressource au sein du système multi-agents. L'interface utilisateur ainsi définie est censée fournir une sorte de bureau électronique offrant un accès uniforme aux applications multi-agents de bureautique communicante. Tous les types de ressources possèdent la même interface de contrôle bas-niveau, et seul l'ensemble des services proposés par l'agent associé est spécifique. L'objet interface propose les trois options évoquées en 7.2.2, à savoir le déménagement vers un autre poste de travail, l'auto-destruction et l'invocation des services accessibles via l'agent.

Comme les agents négociateurs, dont ils sont d'ailleurs les créateurs, les agents salle utilisent tous les modules Prolog. En revanche, les agents utilisateurs ne chargent pas le module définissant la gestion interne de la structure de groupe, qu'ils n'ont pas besoin de connaître. Les bases spécifiques au type de ressource représentée définissent le réflexe de création (objetSMA(init)) afin d'initialiser les valeurs particulières des caractéristiques de l'agent (e.g. identité et fonction de l'utilisateur, surface et équipement de la salle). Ces bases complètent également le comportement et les services particuliers de notre système : réservation de salle, consultation du planning, annulation de réservation…

7.3.2 Prolog et la programmation déclarative

L'implémentation du système de réservation de salle nous a révélé le confort et la puissance de programmation que Prolog apporte à PUMA. Son caractère déclaratif apporte naturellement une grande lisibilité du codage des agents et de leurs interactions (e.g. les messages sont des termes Prolog). De plus, il permet de définir les capacités et les comportements de nos entités de façon :

- modulaire, i.e. en séparant puis combinant différentes catégories de capacités.
- incrémentale, i.e. en déclarant des comportements génériques, spécialisés ensuite.
- dynamique, puisque des éléments de comportement, voire des modules, peuvent être ajoutés en cours de fonctionnement.

Ainsi, le traitement des messages reçus, les services privés ou publics rendus, l'activité, les capacités de coopération et de négociation bénéficient-ils de ces trois propriétés. En particulier, l'aspect dynamique autorise l'extension ou la redéfinition progressive des services rendus par un agent, des messages reconnus, de ses capacités de résolution de contraintes, des structures de coopération utilisables et des stratégies de choix de protocole.

7.3.3 Exemple d'exécution

Le système est implémenté sur des machines de type "PC 486" sous UNIX SCO avec Chorus Fusion et COOLv1. Les objets terminal se basant sur le programme "xterm" pour créer des terminaux de dialogue avec les utilisateurs, le système ne fonctionne que sur X-Window. L'exemple d'exécution qui suit met en jeu trois agents de ressource, répartis sur trois machines (les interventions des utilisateurs sont repérées par des *caractères italiques*).

<u>création d'un agent salle et d'un agent utilisateur</u>

machine SCOOP12, agent bruno	machine SCOOP11, agent dundee
> *runp interface bruno agent_utilisateur* nom : *dillenseger* prenom : *bruno* piece : *171* … groupement/departement : *sce/acs* agent bruno créé Invoquer, Déménager, Quitter ?	> *runp interface dundee agent_salle* piece : *135* groupement : *sce* surface (m2) : *20* … nombre de tables : *5* agent dundee créé Invoquer, Déménager, Quitter ?

Dans les deux cas, on remarque :

(1) la commande de création d'objet COOL/PUMA "runp", avec les arguments "interface" (classe PUMA particulière), le nom symbolique de l'agent, et le nom de fichier Prolog contenant son comportement ;

(2) l'exécution du réflexe de création lié à la classe `objetSMA` (dont hérite la classe `interface`), qui consiste à lire les valeurs des caractéristiques structurelles de l'agent ;

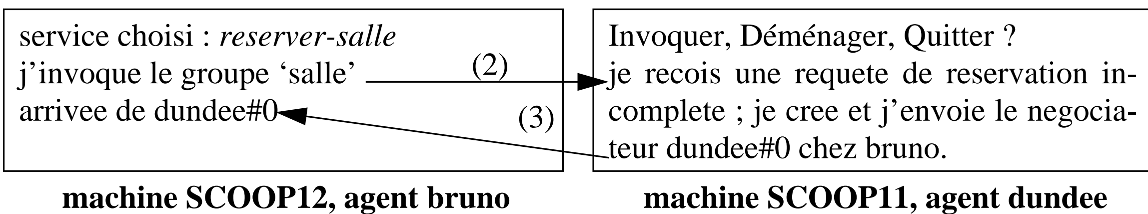
(3) l'affichage du prompt proposant trois commandes à l'utilisateur (ou administrateur de la ressource).

réservation d’une salle

Invoquer, Déménager, Quitter ?*i*
services de l’agent :
* consulter-caracteristiques
* modifier-caracteristiques
* reserver-salle
* annuler-salle
* consulter-memento
service choisi : *reserver-salle*

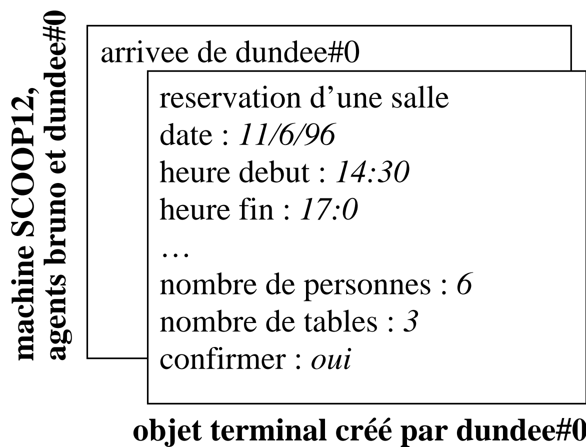
(1) L’invocation de l’agent provoque l’affichage de la liste des services proposés. On y remarque les services d’édition des caractéristiques structurelles de l’agent, communs à tous les agents de ressource.

machine SCOOP12, agent bruno

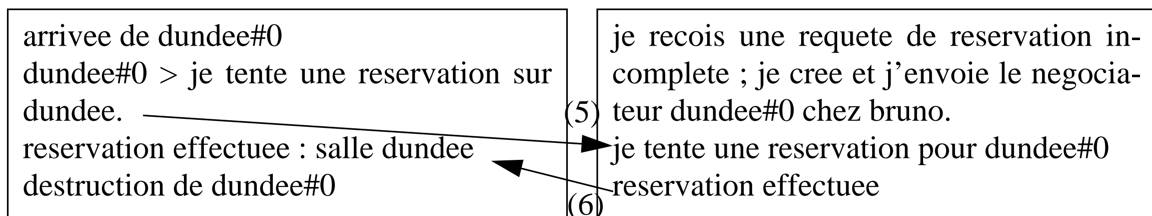


(2) Le groupe des salles est invoqué en mode fonctionnel (mais il n’y a de toute façon qu’un seul membre) par une requête de réservation sans contraintes exprimées.

(3) L’agent Dundee crée un négociateur (dundee#0), qui migre auprès de Bruno.



(4) L’agent négociateur Dundee#0 crée un objet terminal et questionne l’utilisateur pour déterminer les contraintes de sa requête. Dundee#0 interroge également l’agent bruno de façon transparente pour obtenir certaines caractéristiques du client (nom, groupement…).



(5) La salle Dundee étant structurellement compatible avec les contraintes (cf. surface par rapport au nombre de personnes, équipement nécessaire), Dundee#0 lui envoie une requête complète de réservation.

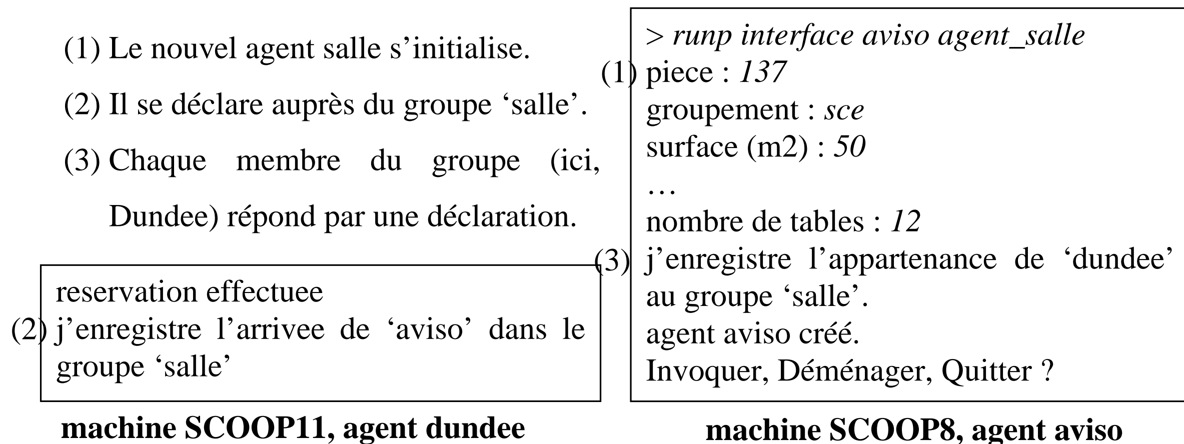
(6) La requête peut être satisfaite par Dundee car elle est compatible avec son planning (il s'agit d'ailleurs de sa première réservation). L'agent bruno en est informé par un rapport de service qui contient tous les paramètres de la réservation. Suite à ce succès, le négociateur Dundee#0 s'auto-détruit.

ajout d'un agent salle supplémentaire

(1) Le nouvel agent salle s'initialise.

(2) Il se déclare auprès du groupe 'salle'.

(3) Chaque membre du groupe (ici, Dundee) répond par une déclaration.



machine SCOOP11, agent dundee **machine SCOOP8, agent aviso**

réservation de salle avec conflit de planning

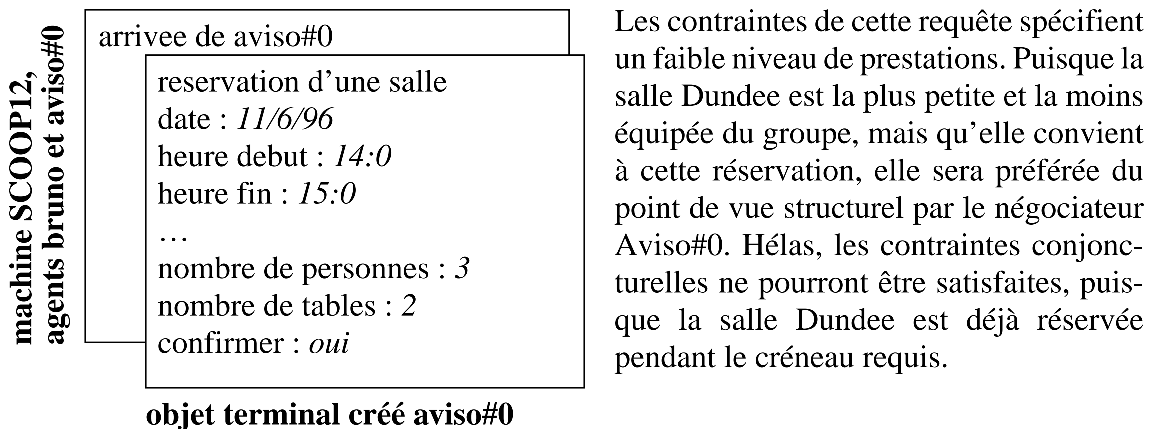
L'utilisateur représenté par l'agent Bruno invoque une seconde fois le service de réservation de salle. Le déroulement est analogue à celui de la première réservation au cours des cinq premières étapes, à la différence près que c'est Aviso qui reçoit la requête incomplète :

(1) invocation de l'agent Bruno ;

(2) envoi d'une requête incomplète vers un membre quelconque du groupe 'salle' ;

(3) création d'un négociateur Aviso#0 par l'agent Aviso, puis migration vers Bruno ;

(4) mise au point interactive de la requête complète ;



objet terminal créé aviso#0

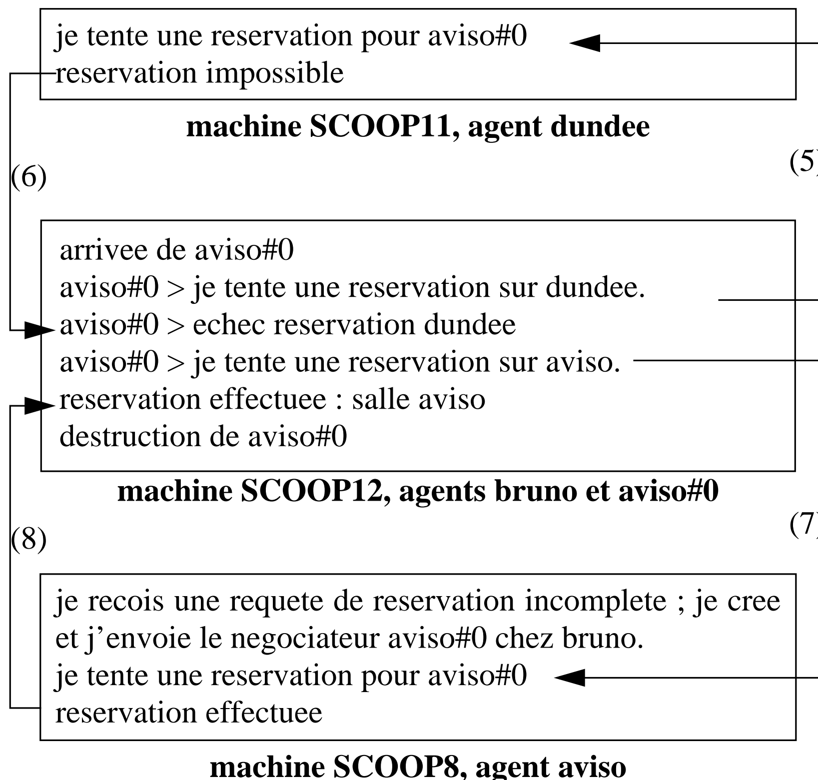
Les contraintes de cette requête spécifient un faible niveau de prestations. Puisque la salle Dundee est la plus petite et la moins équipée du groupe, mais qu'elle convient à cette réservation, elle sera préférée du point de vue structurel par le négociateur Aviso#0. Hélas, les contraintes conjoncturelles ne pourront être satisfaites, puisque la salle Dundee est déjà réservée pendant le créneau requis.

(5) l'agent aviso#0 envoie une requête complète à Dundee ;

(6) la requête échoue pour incompatibilité avec les réservations antérieures ;

(7) l'agent aviso#0 contacte la salle suivante dans sa liste de serveurs potentiels, ordonnée par préférence, soit ici Aviso (qui est d'ailleurs la seule salle restante) ;

(8) la réservation est acceptée par Aviso, l'utilisateur en est informé, puis Aviso#0 s'autodétruit.



7.4 Conclusion

Les deux exemples présentés ont pour objectif de montrer l'intérêt de l'approche multi-agents dans la conception d'applications réparties de bureautique communicante, et d'outils tels que PUMA dans ce contexte.

L'approche multi-agents

En dépit de la limitation des fonctions implémentées, il apparaît que l'approche ouvre la voie à des fonctionnalités avancées et/ou simplifie leur conception ainsi que leur intégration. On peut songer, par exemple, à une renégociation de certaines réservations lorsqu'une annulation intervient ou qu'un agent salle apparaît. De plus, il serait utile de greffer des agents négociateurs “organisateurs de réunion” qui se chargeraient de trouver et réserver les “ressources” nécessaires à une réunion (utilisateurs, salle, équipements). Enfin, moyennant une extension des services rendus, cette couche d'agents peut tout à fait être réutilisée par une autre application multi-agents. Ainsi, dans cet exemple, on peut envisager que les *agents document* d'une application de *Workflow* déclenchent une action de réservation de salle, ou qu'un service de circulation de formulaire soit invoqué par les *agents salle* pour certaines réservations. Les *agents utilisateur* seraient bien entendu les mêmes pour les deux applications.

L'environnement PUMA

En ce qui concerne l'utilisation de PUMA, nous retenons qu'il s'agit d'un outil de maquettage efficace. De plus, ce genre d'outil permet d'envisager de façon naturelle des prolongements en matière de raisonnement et de représentation des connaissances, des compétences, des comportements, de l'activité. Par ailleurs, la modularité et l'aspect incrémental liés à programmation déclarative ne sont pas sans rappeler les propriétés de l'héritage dans le modèle objet, du point de vue de la réutilisation de code et de la spécialisation progressive d'une entité. La notion de conformité de type est présente, car le chargement de certains “modules” permet de conférer des interfaces particulières aux entités (e.g. accès standard aux connaissances d'un agent sur les caractéristiques de ses accointances, services publics). Ainsi, l'association {classe C++ — modules Prolog} proposée par PUMA permet bien de rester dans une approche orientée objet, avec une granularité fixée par les objets COOL/PUMA. De plus, PUMA fournit la possibilité d'un typage dynamique des objets par l'intermédiaire de Prolog. Il semble que toutes ces propriétés (activité autonome, support de

raisonnement, notion de comportement, évolutivité, extension du modèle objet) fassent des objets PUMA de bons candidats pour le label “agent”.

Sur le plan des restrictions que nous devons apporter, il convient de souligner deux faiblesses majeures de PUMA, l’une relevant d’un support insuffisant dans le domaine de la résolution de contraintes, l’autre découlant de l’”obsolescence” de COOLv1. En effet, cette version est désormais abandonnée au profit de COOLv2, basée sur une architecture CORBA, et donc conforme aux standards actuels. Ces constatations se traduisent d’ores-et-déjà par les recherches et réalisations évoquées au Chapitre 6 (autres langages interprétés, intégration sur COOLv2).

Enfin, si l’observation de notre système n’a mis en évidence aucun problème de performance, il est néanmoins possible que des dégradations dues au coût de l’exécution du langage Prolog ou à l’encombrement des objets apparaissent dans un système multi-agents plus large. Aucun véritable test de charge n’a été réalisé, le système étant trop réduit pour être significatif. Cela dit, on note que la représentation des ressources par des agents rend très simple la réalisation de simulations.

Chapitre 8

Structures et protocoles de coopération

8.1 Introduction

Nous nous plaçons ici dans un cadre analogue à celui du système de réservation de salle présenté au Chapitre 7, dont nous allons d'ailleurs décrire la structure et les protocoles de coopération. En particulier, nous supposons qu'un certain degré de sémantique est partagé en ce qui concerne la description d'un service, mais qu'il existe des spécialistes et des non-spécialistes dans chaque domaine de service. Comme nous l'avons déjà exprimé en 4.6 ainsi qu'au Chapitre 5, la mise en relation d'un client avec un serveur "idéal" suppose de surmonter deux difficultés majeures :

- entrer en relation avec les serveurs spécialistes du service recherché,
- être capable de négocier un service pour obtenir l'offre optimale.

Ces deux points sont interdépendants car, d'une part, la recherche des serveurs ad hoc parmi un ensemble d'agents inconnus nécessite l'expression d'un besoin et de compétences, et, d'autre part, la négociation ne peut permettre de choisir le "meilleur" serveur que si l'offre de l'ensemble des serveurs potentiels peut être examinée. Le niveau d'indirection introduit par ce que nous appelons les *structures de coopération* a pour fonction de mettre en correspondance une offre et une demande de façon "optimale". Ces structures s'accompagnent de protocoles dédiés à leur gestion interne et à leur utilisation. Nous présentons en détail l'une d'entre elles,

basée sur la notion de groupe, dont le fonctionnement est adapté à la problématique de notre système de réservation de salle. Enfin, nous suggérons des voies d'approfondissement et de généralisation de la notion de groupe, permettant d'obtenir des structures de coopération "intelligentes".

8.2 Le protocole du "groupe d'artisans"

8.2.1 Objectifs

Dans le système présenté au chapitre précédent, la coopération est basée sur une structure de groupe associée à des protocoles de gestion interne et d'invocation particuliers. Rappelons qu'il s'agit de trouver une salle de réunion adéquate (taille, équipement) et disponible, sachant que chaque salle est représentée de façon exclusive par un agent. Ce contexte induit certaines spécificités concernant les serveurs, les services et les clients du système, dont certaines sont déterminantes dans la conception de notre structure de groupe :

- l'apparition ou la disparition d'un serveur est un événement très rare ;
- le nombre de ces serveurs est plutôt réduit ;
- on distingue des caractéristiques structurelles (surface, emplacement, certains équipements[1]) et des caractéristiques conjoncturelles (planning des réservations) ;
- il y a potentiellement beaucoup de clients ;
- a priori, les clients ne connaissent ni les serveurs, ni les caractéristiques de service à négocier.

Les protocoles mis en œuvre pour la gestion et l'invocation du groupe sont adaptés à ces propriétés, et ils ont été conçus dans l'optique de réduire les inconvénients de protocoles tels que le réseau de contrat qui, par des flux importants de messages, engendrent :

- la surcharge du réseau de communication,
- l'encombrement des boîtes aux lettres des agents,
- le ralentissement de l'activité des agents dû au traitement des messages reçus.

De plus, nous avons voulu prendre le parti d'une répartition totale et symétrique de l'organisation des serveurs, afin d'éliminer toute trace d'instance centrale.

1. Certains équipements, de par leur mobilité, peuvent être classés dans les deux catégories, ou non pris en compte par le système. Voir la discussion à ce sujet en 8.3.2.

8.2.2 Principe

La structure mise en place consiste à ce que tous les agents salle maintiennent une sorte d'association d'artisans d'un même domaine, dont le but n'est pas la concurrence mais la coopération visant la satisfaction optimale des clients. Ce groupe de serveurs s'enregistre sous un nom symbolique attaché à une sémantique partagée par les tous les agents du système susceptibles d'être clients. Ainsi le groupe “salle” rassemble un type d'agent particulier offrant les services de réservation de salle, d'annulation de réservation, et de consultation du planning. Le groupe introduit un niveau d'indirection qui permet aux clients de s'affranchir du nom d'un serveur particulier, et leur offre l'opportunité de contacter plusieurs serveurs (en fait, tous les serveurs correspondant à un certain type de ressource) à travers une seule adresse.

Notre organisation particulière de groupe s'appuie largement sur les mécanismes de groupe offerts par Chorus, qu'il s'agisse de la constitution d'un groupe ou de son invocation. Ceci nous soustrait des préoccupations concernant la création d'une adresse de groupe, de l'ajout ou du retrait d'un membre, ainsi que de la diffusion globale ou fonctionnelle d'un message sur un groupe (i.e. tous les membres du groupe ou un membre aléatoire). En revanche, nous construisons des protocoles de gestion interne et de coopération au-dessus de ces fonctionnalités.

8.2.3 La gestion interne du groupe

Lorsqu'un agent salle est créé dans le système, il diffuse un message au groupe des salles pour déclarer son apparition. En retour, chaque membre du groupe répond par un message de déclaration d'appartenance à ce groupe[2]. Comme ces messages de déclaration contiennent les caractéristiques structurelles (surface, équipement, emplacement...) de l'agent émetteur, il en résulte que chaque salle connaît en permanence toutes les autres salles et leurs caractéristiques structurelles. Les caractéristiques conjoncturelles (e.g. planning) ne sont pas transmises afin de ne pas abuser de la diffusion de groupe. De façon inverse, tout agent se retirant d'un groupe diffuse un message spécifique pour en informer les autres membres (cf. figure 34).

2. Signalons qu'un protocole similaire est exploité par le micro-noyau Chorus lors de l'apparition (boot) d'une machine sur le réseau, afin de déterminer son “numéro d'incarnation” indispensable à la génération d'identificateurs uniques (“hello protocol”).

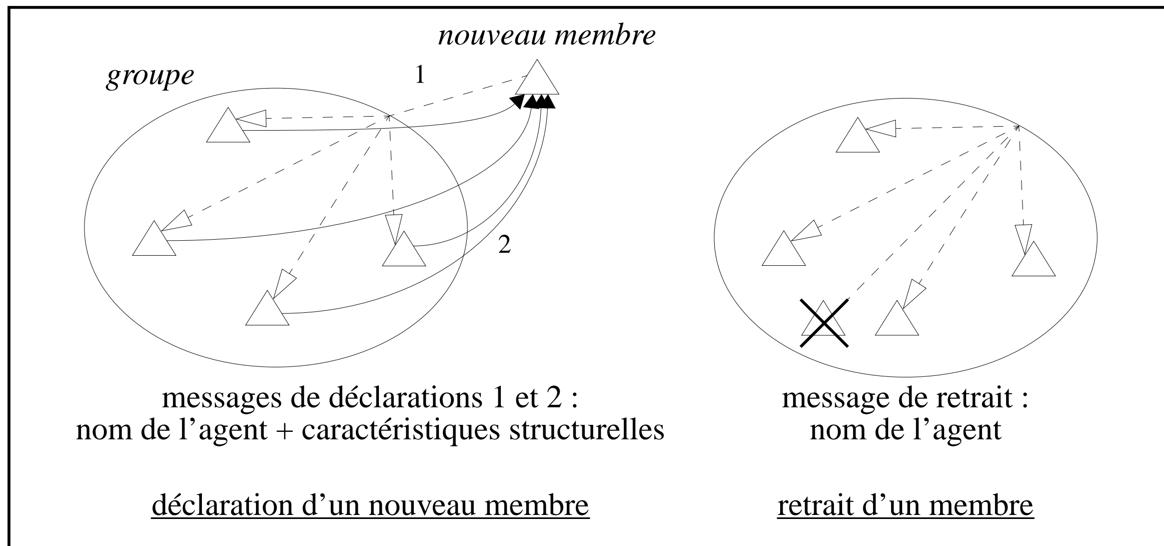


figure 34 : gestion interne de la structure de groupe

Le bon fonctionnement de cette structure repose sur deux hypothèses, qui expriment le fait que les communications dues à la gestion interne du groupe doivent être négligeables par rapport aux invocations de services :

(1) le groupe est lentement dynamique, i.e. la création, la disparition d'un membre ou l'évolution de ses caractéristiques structurelles sont des phénomènes rares ;

(2) le nombre de membres est réduit.

En effet, lorsqu'un groupe comporte n membres, la déclaration d'un nouveau membre provoque l'émission de 2n messages, dont n dans une même diffusion. Même si le coût de diffusion est difficilement estimable a priori puisque nous ne présumons pas de l'implémentation des mécanismes système, on notera quand même que n messages sont envoyés individuellement, qui plus est à destination de la même boîte aux lettres[3]. D'un point de vue général, puisque nous procédons à des duplications d'information, les deux hypothèses permettent de considérer qu'il n'est pas trop coûteux de maintenir un niveau de cohérence satisfaisant.

3. Sur le principe, une amélioration (proposée dans [MAL 96]) pourrait être apportée en envoyant le message de déclaration à un membre aléatoire du groupe (mode fonctionnel), qui diffuserait cette déclaration à ses collègues et répondrait au nouvel arrivant par un unique message contenant les caractéristiques de tous les membres du groupe. Ceci permettrait de réduire le nombre de messages engendrés à n+2, mais en pratique, la taille importante du message de réponse envoyé au nouveau membre peut éventuellement poser problème (selon le nombre de membres et de caractéristiques structurelles), le volume total d'information transmise restant le même. En outre, le traitement d'un gros message accroît la durée de l'"étape élémentaire" de l'activité de l'agent, par rapport à un fractionnement du message en n morceaux.

8.2.4 L'invocation du groupe

Puisque les membres se connaissent les uns les autres, il est inutile de diffuser une requête de service au groupe entier. Lorsqu'un agent utilisateur désire réserver une salle, il lui suffit de contacter un seul agent salle. Comme le client n'est pas supposé connaître de serveur a priori, il invoque le groupe de communication en utilisant le mode *fonctionnel*, qui consiste à envoyer un seul message à un membre quelconque du groupe.

Cas d'une requête "complète" (figure 35)

Si l'agent client est capable de formuler une requête de service complète, i.e. spécifiant des contraintes sur les caractéristiques propres au serveur et au service, celle-ci est confrontée aux caractéristiques structurelles de l'ensemble des membres par le récepteur du message. Un filtrage des serveurs structurellement incompatibles avec la requête est alors opéré, puis la liste des serveurs potentiels est ordonnée par ordre de préférence. Ce classement est effectué à partir du critère de satisfaction éventuellement exprimé par le client, ou, à défaut, en appliquant un critère propre à l'ensemble des serveurs. Ainsi, dans notre système de réservation, l'agent salle intermédiaire cherche à favoriser les salles dont l'équipement est juste suffisant.

Ensuite, l'agent intermédiaire contacte successivement les serveurs de cette liste tant qu'il reçoit des refus pour cause d'indisponibilité, ce qui correspond à une incompatibilité concernant les caractéristiques conjoncturelles du serveur candidat. Lorsqu'un serveur est capable de traiter la requête, il l'effectue et envoie un contrat de service au client. Si aucun serveur ne convient, le client reçoit un avis d'échec de la part de l'agent intermédiaire.

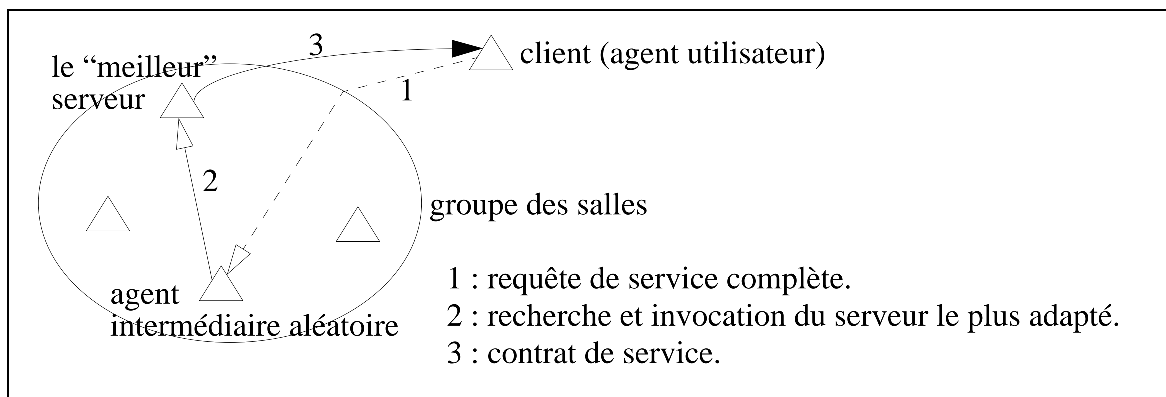


figure 35 : invocation de la structure de groupe par une requête complète

Cas d'une requête incomplète (figure 36)

L'agent utilisateur n'étant pas un expert dans le domaine des réservations de salle, il est légitime de supposer qu'il soit incapable de formuler une requête de service complète. Par conséquent, l'agent intermédiaire qui reçoit l'invocation répond en créant un agent auxiliaire, sorte d'*agent négociateur*, chargé de s'occuper de la requête du client. L'agent négociateur recopie la liste des agents salle ainsi que leurs caractéristiques structurelles, auprès de son créateur. Ensuite, il migre vers le site du client et établit de façon interactive la requête de service complète. Il interroge l'utilisateur pour déterminer les principales contraintes, mais il accède aussi directement à son agent pour récupérer certaines informations. Ainsi, la répartition logique et physique du problème est respectée : la requête de service complète est établie par un expert du domaine en provenance d'un site distant.

A partir de l'instant où une requête complète peut être formulée par l'agent négociateur, la suite des opérations est analogue au cas précédent, l'agent négociateur jouant le rôle de l'agent intermédiaire : classement des serveurs structurellement compatibles, recherche séquentielle d'un serveur disponible, invocation du service et envoi du contrat ou de l'avis échec.

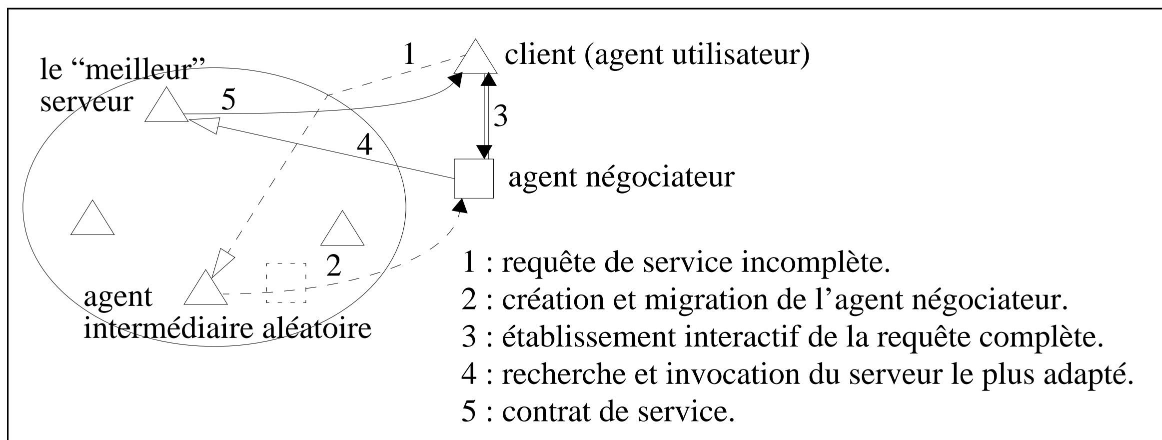


figure 36 : invocation de la structure de groupe par une requête incomplète

8.2.5 Evaluation

Notre protocole de recherche du "meilleur" serveur est clairement optimisé : peu de messages, étalés dans le temps et sans diffusion. S'il est vrai que la migration de l'agent négociateur est une opération coûteuse, il faut néanmoins souligner qu'elle n'a lieu qu'une fois et qu'elle possèdent des vertus essentielles. En premier lieu, elle permet d'éviter les nombreuses communications qui auraient été nécessaires pour dialoguer avec l'utilisateur à distance. En second lieu, la co-résidence du négociateur et du client permet des interactions rapides, non tributaires des performances du réseau de communication (voir la notion de

“Remote Programming” en 4.4.4). Par ailleurs, certains échanges d’information locaux entre agents sont réalisés par invocation synchrone et non par messages. L’accès direct au noyau Prolog d’un autre agent est un mécanisme efficace, qui simplifie et allège beaucoup la conception de ces transferts : pas de gestion de messages et transparence de l’accès, tant du point de vue de l’agent cible que de l’agent appelant[4].

A titre d’illustration, le coût de recherche d’un service en terme de messages est comparé avec le protocole du réseau de contrat (CNet) dans le tableau 37, n représentant le nombre de serveurs potentiels. Bien sûr, cette comparaison n’est valable que dans le cadre des hypothèses pré-citées qui permettent de négliger les communications ponctuelles dues à la gestion interne du groupe. Une caractéristique intéressante de notre structure est de permettre une estimation partielle des offres a priori, puis une recherche du meilleur serveur de façon séquentielle et ordonnée, ce qui évite le dilemme suivant :

- soit le serveur candidat garantit son offre en réservant les ressources nécessaires, mais il risque alors de décliner à tort d’autres appels d’offres s’il n’est finalement pas élu ;
- soit le serveur ne réserve pas les ressources nécessaires, auquel cas son offre est instable.

	réseau de contrat		groupe	
au mieux	- n appels d’offres - 1 offre - 1 contrat - 1 rapport d’exécution	**n+3 messages**	- 1 requête non ciblée - 1 requête ciblée - 1 rapport d’exécution	**3 messages**
au pire	- n appels d’offres - n offres - n contrats - n rejets[a]	**4n messages**	- 1 requête non ciblée - n requêtes ciblées - n rejets	**2n+1 messages**
cas “moyen”	- n appels d’offres - n/2 offres - n/4 contrats - n/4 messages (rejets + 1 rapport)	**2n messages**	- 1 requête non ciblée - n/4 requêtes ciblées - n/4 messages (rejets + 1 rapport)	**n/2+1 messages**

tableau 37 : évaluation et comparaison du réseau de contrat et du protocole de groupe

a. Entre le moment où le serveur établit son offre et où il reçoit le contrat, il est possible qu’il ne puisse plus l’honorer, suite à de nouveaux contrats acceptés.

4. Du point de vue des systèmes multi-agents, et en adoptant une vision anthropomorphique, on pourrait rapprocher ce mode de communication synchrone avec la notion de perception d’autrui. En effet, la communication s’effectue de façon locale, sans participation active de la part de l’agent observé, et l’agent observateur n’accède en fait qu’à une apparence de l’autre (données particulières mises à disposition par chaque agent, accédées de façon “normalisée” au sein du système).

8.3 Réflexions

8.3.1 Portée de la notion de groupe

La gestion "système" des groupes

On assiste actuellement à de nombreux développements liés à la notion de groupe. Ainsi, au niveau des réseaux, les modes "unicast" (point à point) et "broadcast" (diffusion générale) du protocole IP se voient désormais complétés par des adresses "anyone" et "multicast", qui correspondent aux deux variantes de diffusion de groupe (un membre quelconque ou tous les membres). Les systèmes d'exploitation prennent également en compte cette notion de groupe, avec par exemple la gestion de groupes de processus dans ISIS. En ce qui concerne Chorus, micro-noyau adapté à la réalisation de systèmes répartis, la communication et la gestion de groupes de communication sont au cœur de l'architecture. Enfin, le langage C++ lui-même fait l'objet de propositions de mécanismes de groupe.

A l'heure de l'extension rapide et de l'interconnexion des réseaux, le principe de diffusion global n'est plus viable, tant du point de vue des réseaux d'acheminement que des capacités de traitement des machines. Pourtant, le besoin d'ouverture des systèmes impose davantage de recherches d'entités a priori inconnues. Dans ce contexte, le degré d'indirection offert par les groupes permet des communications ciblées, sous réserve que des protocoles de gestion et d'utilisation particuliers soient définis. Ces protocoles déterminent alors l'équilibre entre le degré de filtrage offert par les groupes et le niveau de connaissance pré-requis pour leur utilisation.

Mise en œuvre dans des protocoles de niveau supérieur

Nous abordons les mécanismes "bas niveau" de gestion et de communication de groupes comme relevant du niveau système. Pourtant, ceci ne signifie pas que ces fonctionnalités ne justifient pas des recherches et des développements propres (bien au contraire), mais simplement qu'elles ne constituent pas en elles-mêmes l'objet de nos travaux. Ainsi, notre protocole de groupe ne présume pas de leur implémentation au sein du micro-noyau Chorus et de la couche orientée objets COOL. En revanche, nous les appliquons à la construction de structures de coopération de plus haut niveau, allant dans le sens d'une limitation du nombre de messages échangés. La mise en oeuvre des groupes devant apporter une rationalisation des

communications, l'optimisation du nombre d'entités membres d'un groupe est un point clé de la gestion interne du groupe.

En effet, l'efficacité d'un groupe en terme d'indirection (i.e. nombre de serveurs potentiellement contactés via l'adresse de groupe) impose que le nombre de membres soit significatif. Des groupes trop petits aboutissent à une multiplication du nombre de groupes au sein du système et traduisent un morcellement des types de compétence. Ceci impose aux clients potentiels de connaître beaucoup de groupes, dont la caractérisation en terme de type de service doit être très précise. A l'inverse, un nombre trop élevé de membres est néfaste à l'efficacité du groupe puisque sa gestion interne et/ou son invocation (selon les protocoles définis) se confronte au problème des diffusions massives. Certes, nous avons admis l'existence de services système de communication de groupe efficaces et transparents. Mais, de même qu'admettre la disponibilité d'un réseau viable résulte toujours de façon plus ou moins implicite d'une hypothèse sur la largeur de bande passante nécessaire, il est évident que ces services de groupe ont des limites en termes de nombre de membres et d'échelle du réseau sur lequel ils se répartissent.

Groupe de groupes et généralisation

Nous avons présenté un système dans lequel les serveurs “artisans” d'un domaine s'organisent dans un groupe qui doit être connu d'avance par les clients. Ce niveau unique d'indirection exige une connaissance précise de la part des clients sur l'existence des groupes et les services qu'ils proposent. De plus, aucune dynamique d'apparition ou de disparition de groupe n'est prévue. Or, comme nous l'avons signalé, le nombre de membres d'un groupe est déterminant pour son bon fonctionnement. Si les membres sont trop nombreux, il peut s'avérer nécessaire de scinder un groupe en deux sous-groupes davantage spécialisés, non nécessairement disjoints. Inversement, des groupes embryonnaires n'ont pas de raison d'être individuellement mais peuvent former un groupe cohérent plus général.

Par exemple, on peut envisager qu'un groupe des imprimantes donne lieu à deux types de sous-groupes :

- un groupe des imprimantes noir et blanc complémentaire d'un groupe des imprimantes couleur ;
- un groupe des imprimantes locales complémentaire d'un groupe des imprimantes distantes.

Par itération de ce principe, on voit qu'il apparaît une notion de chemin d'accès non unique à un serveur. On peut envisager qu'un groupe “racine” subsiste dans un état vestigiel afin de rediriger les recherches de serveur vers les nouveaux groupes, connaissances qui pourrait être acquises par les clients afin d'éviter qu'un goulet d'étranglement ne se forme.

De nouveaux protocoles de recherche de serveur pourraient être également envisagés si l'on disposait de mécanismes de groupe permettant de réaliser des opérations ensemblistes. Par exemple, l'intersection de plusieurs groupes sémantiquement non disjoints, tels que dans l'exemple des groupes d'imprimantes spécialisés, permettrait de cibler davantage la recherche.

8.3.2 Influence du “type” de caractéristique et du “type” de service

Caractéristique structurelle ou conjoncturelle ?

La distinction entre caractéristique structurelle ou conjoncturelle n'est pas toujours immédiate. Ainsi, dans l'exemple du système de réservation de salle, la surface est directement liée à la conception du serveur, alors que le planning des réservations relève de l'utilisation de la ressource à un instant donné. En revanche, le nombre de chaises peut être considéré comme définissant le serveur ou représentant un état particulier. Le choix résulte non seulement d'une règle d'organisation (les chaises peuvent/doivent-elles être déplacées ?), mais également d'un soucis d'efficacité de notre structure de groupe. On voit donc que la notion de caractéristique structurelle ou conjoncturelle peut apparaître subjective, mais que cette subjectivité se traduit par la possibilité d'ajuster le fonctionnement du groupe selon :

- la fréquence des mises à jour de la caractéristique,
- la performance du filtrage associé à la caractéristique,
- l'ordre de grandeur du nombre de membres du groupe.

Spécialisation du service

Par ailleurs, la notion de caractéristique structurelle appelle des commentaires particuliers vis-à-vis du principe de l'association d'un groupe à un “type” de service. En effet, dans la mesure où cette caractéristique définit la structure de la ressource, elle spécialise le “type” de service rendu. On voit alors qu'à l'ajustement entre caractéristique structurelle et conjoncturelle s'ajoute un réglage de la précision du type de service associé à un groupe. Ce deuxième paramètre permet de conditionner le nombre de membres d'un groupe, conformément aux préoccupations présentées en 8.3.1.

8.3.3 La question des réseaux à grande échelle

Si l'approche multi-agents du système d'information et le principe de structures de coopération présentent des contours assez clairs à l'échelle d'une entité administrative, nous devons pourtant signaler une question que nous n'avons pas abordée : celle de l'interconnexion de plusieurs "domaines". Deux problèmes se posent alors, l'un relevant du volume énorme d'information dû à l'échelle du système résultant, l'autre de l'apparition d'une hétérogénéité sémantique. De façon parallèle, ces deux points peuvent être également considérés en terme de bande passante et d'hétérogénéité du point de vue des réseaux et des systèmes d'exploitation.

En ce qui concerne les couches basses, les travaux dans les domaines des normes et standards permettent progressivement d'augmenter le niveau d'interopérabilité. Ceci ne relève pas de nos travaux, mais il est clair que les notions de LAN, MAN et WAN[5] traduisent l'importance de l'impact de l'échelle d'un réseau sur son fonctionnement. Dans le cas de Chorus, par exemple, certains mécanismes sont opérationnels au sein d'un *domaine*, réunissant un nombre limité de machines. Faudra-t-il développer des protocoles s'adaptant à l'échelle du réseau, des structures d'interconnexion (e.g. des services objets particulier dans CORBA) ? La large bande passante des futures autoroutes de l'information rendra-t-elle caduques certaines problématiques ?

Pour notre approche multi-agents, en l'état actuel des moyens informatiques, une approche viable consisterait à créer des agents particuliers (un groupe d'agents ?) au sein de chaque entité administrative, chargés de permettre la coopération avec des entités extérieures. Ces agents fourniraient une structure de coopération tournée vers l'extérieur, assurant notamment un rôle de traduction entre domaines hétérogènes, mais également de protection, d'adaptation du nommage…

8.4 De la structure de groupe à l'annuaire intelligent

8.4.1 Limites de la structure de groupe présentée

Lorsque nous avons présenté notre structure de coopération basée sur le principe du groupe de serveurs, nous avons d'une part précisé deux hypothèses (nombre réduit de membres et stabilité de la composition du groupe) et d'autre part, exploité les spécificités du cadre applicatif. En effet, notre structure est adaptée à la recherche d'agents dont le "type" (i.e. les

5. Local/Metropolitan/Wide Area Network.

services offerts) est un critère particulièrement discriminant. Il est alors légitime, immédiat et très efficace de les associer dans un groupe, représenté par une adresse symbolique connue des agents clients potentiels. De plus, les caractéristiques que l'on peut rattacher à des serveurs de type "salle" sont en nombre restreint. Ceci favorise la stabilité et limite le volume global des données répliquées (i.e. les caractéristiques structurelles). Dans un contexte de bureautique communicante, la structure de groupe associée à la vision multi-agents permet d'intégrer simplement et efficacement de nombreuses ressources : groupe des imprimantes, des télécopieurs, des serveurs "intranet", des véhicules de service, etc.

Mais il peut être nécessaire de rechercher une ressource caractérisée autrement que par le type de service rendu, notamment dans la mesure où ce critère se révèle peu discriminant. De plus, il nous faut aborder le cas d'un grand nombre de ressources similaires. Ainsi, il est inconcevable de gérer un groupe des utilisateurs au sein d'une entreprise, car le nombre potentiellement très important de membres et de caractéristiques associées rendrait la structure ingérable. En outre, la grande variété des caractéristiques empêche d'envisager la création de groupes d'utilisateurs plus spécialisés. En effet, cela engendrerait une multitude de groupes non disjoints, d'où une multiplication des communications (cf. gestion interne de chaque groupe dans lequel l'utilisateur est impliqué) et des connaissances nécessaires sur les groupes de la part des agents du système. Par exemple, dans le cas d'une messagerie intelligente ou de la circulation intelligente de document, il serait utile de pouvoir retrouver une personne correspondant à un profil particulier (e.g. possédant une voiture rouge, ou ayant des compétences/responsabilités particulières dans un projet).

8.4.2 X.500 et la notion d'annuaire

Les annuaires constituent des types particuliers de bases de données dont l'utilisation correspond à notre problématique de recherche d'agents. Ainsi, la norme d'annuaire réparti X.500 peut être vue comme la description d'une structure de coopération. D'ailleurs, les "courtiers" de certains systèmes répartis à objets implémentent cette norme pour représenter les serveurs et les services qu'ils exportent.

Présentation de la norme X.500

Un annuaire X.500 représente un espace de nommage universel ("Directory Information Base") regroupant une collection d'objets organisés de façon arborescente ("Directory Information Tree"). Chaque objet contient un ensemble d'attributs et peut être identifié de

façon relative à une position dans l'arbre, ou par un nom unique ("Distinguished Name") correspondant au chemin complet depuis la racine du DIT. Cet arbre est réparti sur plusieurs serveurs ("Directory System Agent") responsables de portions disjointes. Les DSA coopèrent via un protocole de gestion interne ("Directory System Protocol") et rendent un service d'annuaire accessible par le biais d'un protocole d'accès ("Directory Access Protocol") et d'une entité applicative ("Directory User Agent").

Les points forts

Bien que son implémentation soit parfois considérée comme complexe et donc coûteuse, l'annuaire X.500 possède de nombreux atouts. Outre sa prise en compte des questions d'authentification et de contrôle d'accès, il est particulièrement adapté à la définition d'annuaires d'entreprise par son aptitude à répertorier indifféremment les entités administratives et hiérarchiques, les responsabilités, les personnes, et les serveurs informatiques. De plus, le principe de la représentation répartie du DIT par plusieurs DSA interopérant permet d'envisager une interconnexion à grande échelle d'annuaires. Ainsi, la spécification de schémas standard et d'un protocole d'invocation simplifié ("Lightweight DAP") est en passe de déclencher son développement sur Internet pour offrir un service réparti de nommage mondial.

Les limites

Certaines limitations de l'annuaire X.500 résultent souvent d'utilisations abusives de type base de données pour stocker des informations insuffisamment stables. En effet, la norme X.500 précise que les opérations de modification doivent être des événements sporadiques par rapport aux interrogations. Une expérience de réalisation d'un système de réservation basé sur X.500 a ainsi montré que l'annuaire était confronté à une problématique du type caractéristique structurelle / caractéristique conjoncturelle, ayant conduit à l'utilisation d'une base de données externe pour gérer les emplois du temps, trop sujets à variation. Les problèmes de performance découlent également de l'absence de redondance, puisque les DSA gèrent des portions disjointes du DIT. Certes, la notion de réplication apparaît dans X.500 version 1992, mais avec un faible niveau de cohérence. Enfin, la logique arborescente de l'organisation des objets et la rigidité du schéma ne siéent pas à tout type d'application nécessitant un service de nommage.

Par exemple, la réalisation pratique du module CIDRIA du projet PREVISIA a révélé les difficultés d'utiliser X.500 pour répertorier toutes les données nécessaires à une application de Workflow. La multiplication des rôles bureautiques liés à la structure hiérarchique, aux compétences, aux projets, aux responsabilités, rendent très complexe le schéma de l'annuaire. Il est difficile de définir celui-ci de façon standard, y compris au sein d'une unique entreprise, et, en dépit d'une organisation hiérarchique, la structure arborescente n'est pas adaptée. De plus, la prise en compte des délégations et des absences compliquent son administration et nuisent à la stabilité des attributs répertoriés.

8.4.3 Un groupe d'agents "annuaire intelligent"

Au lieu de fédérer tous les utilisateurs dans un groupe, nous proposons de créer un groupe de serveurs de type annuaire. Chaque membre de ce groupe est chargé de répertorier l'ensemble des caractéristiques structurelles d'utilisateurs vérifiant certaines contraintes. Ces contraintes forment le "filtre", caractéristique structurelle propre à chaque agent annuaire. Il peut y avoir une intersection non nulle entre deux filtres (cf. figure 38), mais la réunion de l'ensemble des filtres doit être le filtre nul (i.e. aucune contrainte). Par ailleurs, la cohérence du service d'annuaire rendu par ce groupe suppose que chaque agent utilisateur se déclare au moment de sa création et signale ensuite toute modification de ses caractéristiques structurelles, ainsi que son éventuelle disparition. Pour cela, un message doit être diffusé au groupe des annuaires. En revanche, l'invocation du groupe se fait en mode fonctionnel, conformément au protocole décrit précédemment. En effet, l'agent annuaire qui reçoit la requête de recherche peut la rediriger sur son ou ses "collègues" dont le filtre est compatible avec les contraintes définissant l'utilisateur recherché.

Si la structure de groupe en elle-même est principalement l'occasion de mettre en évidence les bénéfices tirés du système réparti à objets sous-jacent (cf. groupes de communication, migration) dans notre contexte multi-agents, son application à un service d'annuaire ouvre la voie à une véritable exploitation de capacités de raisonnement. En effet, la configuration des différents agents qui le composent peut se faire dynamiquement et de façon intelligente, selon des processus typiques de l'intelligence artificielle. Supposons que l'on parte de la configuration minimale consistant en un unique agent répertoriant tous les utilisateurs (i.e. filtre nul). Connaissant toutes les caractéristiques des utilisateurs, il peut appliquer un processus de taxinomie afin de se diviser en plusieurs parties spécialisées (création de filtres "complémentaires"). Inversement, des agents annuaires peuvent se

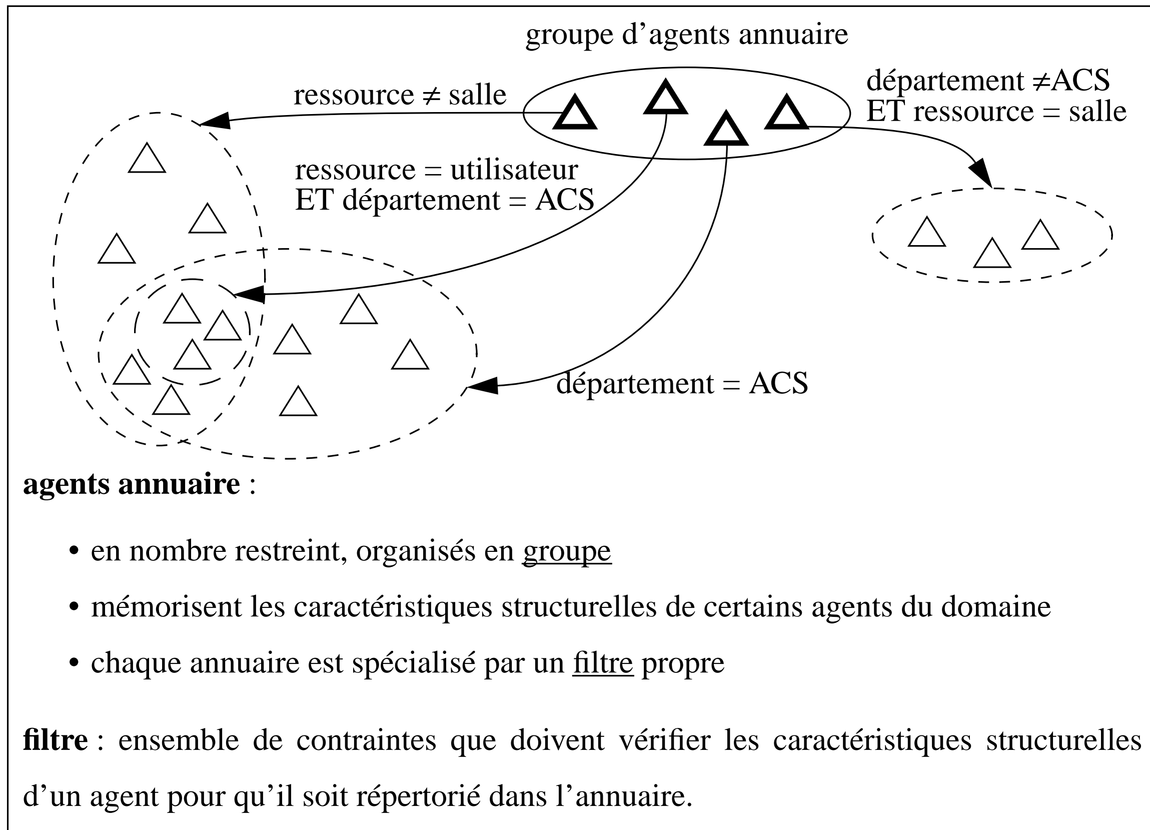


figure 38 : illustration du principe de groupe d’agents annuaires

regrouper s’ils s’avèrent être trop spécialisés. En fonction de son taux de sollicitation, un agent annuaire peut également décider de se dupliquer ou de fusionner avec un autre.

Enfin, pour une optimisation complète du service d’annuaire, cette configuration logique définie par les filtres des agents doit être liée à une configuration physique. Le concept de placement dynamique d’objets dans un système réparti, consistant à corréler l’intensité des relations logiques (interactions) des entités avec leur localisation relève des travaux présentés dans [BOU 93].

8.5 Conclusion

Comment optimiser la recherche d’un service et d’un serveur par un client, ou comment favoriser la mise en relation d’agents qui ont besoin de coopérer dans un environnement réparti, ouvert et dynamique ? Les problématiques, les arguments et les exemples relatifs à la notion de structure de coopération et de protocoles associés, présentés dans ce chapitre, montrent qu’il s’agit d’un vaste domaine. Notre objectif était donc de mettre en évidence ce sujet de recherche à part entière, qui fait d’ailleurs l’objet d’une thèse en cours au SEPT.

Nos propos sont illustrés par la mise en œuvre d'une structure de coopération basée sur la notion de groupe de serveurs, et une proposition de structure de type annuaire construite au-dessus de cette structure. Il s'en dégage un certain nombre d'idées fortes :

(1) Le système doit comporter plusieurs structures différentes. La multiplicité des situations ne permet probablement pas de définir une structure universelle répondant de façon optimale à tous les besoins de coopération.

(2) L'optimisation de ces structures passe par une prise en considération de la dynamique faible ou forte des caractéristiques des agents du système.

(3) La notion de groupe de communication ouvre des perspectives très intéressantes vis-à-vis de nos préoccupations.

(4) L'émergence et l'auto-régulation de ces structures nécessitent des capacités de raisonnement et des techniques relevant de l'IA.

(5) L'optimisation du fonctionnement global du système passe par des capacités de mobilité des agents et des fonctionnalités de placement dynamique "intelligent".

Chapitre 9

Conclusion

9.1 Bilan

Nos travaux se situent au croisement de trois domaines :

- la bureautique communicante,
- les Systèmes Répartis à Objets (SRO),
- les Systèmes Multi-Agents (SMA).

A défaut d'être exhaustif, il nous a paru important d'en proposer un point de vue aussi synthétique que possible, tout en précisant certains détails nécessaires à la bonne compréhension des tenants et des aboutissants de notre approche.

Ainsi, nous avons en premier lieu exposé les problématiques liées au système d'information de l'entreprise, et plus particulièrement aux applications de bureautiques communicantes. Nous avons insisté sur les systèmes de *Workflow*, dont l'analyse des besoins constitue l'origine de nos travaux. Ensuite, nous avons présenté l'approche des systèmes répartis à objets, et notamment l'architecture CORBA de l'OMG. Ce type de support est déterminant pour la prise en compte de la répartition naturelle, tant géographique que fonctionnelle, de notre contexte applicatif. En particulier, nous avons détaillé le système COOL, initialement spécifié par le SEPT pour répondre à des besoins de bureautique communicante, et qui sert de support technique à notre implémentation. Enfin, nous nous

sommes intéressé à l'Intelligence Artificielle Distribuée et aux SMA. Ce rapprochement était naturel en raison de la mise en évidence d'un besoin de raisonnement réparti, tant au niveau des applications que des fonctionnalités "système". La notion d'agent se révélant très en vogue, et ce parmi diverses communautés techniques et scientifiques, il nous a paru nécessaire d'en rechercher les propriétés marquantes vis-à-vis de nos préoccupations.

Suite à ce "triptyque", nous avons proposé une approche multi-agents assez générale des systèmes de bureautique communicante. Nous avons cherché à montrer les potentialités de ce "modèle" en matière de coopération optimale, de négociation intelligente, dans un environnement ouvert de ressources hétérogènes et évolutives.

Cette approche multi-agents donne lieu à une implémentation sur système réparti à objets, par l'intermédiaire de notre système de développement d'agents PUMA. Le Prolog pour Univers Multi-Agents résulte de l'intégration d'un interpréteur Prolog au sein d'un objet C++ sur le système COOL. Il en découle un certain nombre de classes d'objets offrant une multitude d'avantages, notamment en termes de prototypage rapide d'application, de support de raisonnement, de typage dynamique, mais également de migration avec conservation de l'activité en cours. La portée de ce type d'outil est réellement au cœur de nos travaux :

- il fournit un outil d'implémentation de SMA, dont la répartition physique est effective ;
- vis-vis des SRO et des applications qu'ils supportent, il offre l'opportunité d'intégrer des techniques d'IAD.

Dans le cadre d'une application simple, la réalisation pratique d'agents coopérants représentant les ressources d'un système nous a permis de mettre en évidence le besoin de mécanismes "intelligents" pour mettre en relation de façon efficace un agent client avec l'agent serveur "idéal". Cette intelligence doit s'exprimer au niveau de la négociation et au travers de structures de coopération à la configuration dynamique et de protocoles associés.

Comme l'illustrent les perspectives ci-après, certains principes clés mis en avant dans cette thèse sont actuellement repris pour être redéveloppés, adaptés et approfondis sur des technologies plus récentes :

- le support de communication déterminant apporté par le SRO sous-jacent ;
- l'utilisation de langages interprétés dans un objectif de programmation par script et/ou de résolution de problème et de représentation des connaissances ;
- le nomadisme ;

- les structures et les protocoles de coopération ;
- le modèle de coopération multi-agents sur SRO.

On se félicitera des fonctionnalités offertes par l'intégration d'un interpréteur Prolog au sein d'objets C++ sur système réparti, et de l'utilisation des classes résultantes pour implémenter un modèle d'agent coopérant, actif et nomade. On remarquera un début d'introduction de techniques de type IA/IAD dans les SRO pour résoudre des problèmes liés aux applications réparties. On regrettera que le temps nous ait manqué pour réaliser une véritable maquette de circulation de documents ou d'autres applications de travail coopératif. De même, certaines idées concernant les structures et les protocoles de coopération n'ont pas pu être évaluées. En fait, ces deux restrictions traduisent les lacunes de notre plate-forme en termes d'outils d'observation et d'interface utilisateur.

9.2 Perspectives

Alors que nous avancions dans nos travaux, nous avons observé l'émergence des systèmes répartis à objets, désormais standardisés par l'architecture CORBA. De plus, nous avons découvert plusieurs systèmes de recherche, dans le domaine de la bureautique communicante, qui adoptaient une démarche “orientée agent”. Hier dans l'air du temps, ces idées sont aujourd'hui très à la mode, particulièrement en ce qui concerne le concept d'agent.

L'originalité de notre approche est sans doute d'avoir tenté, dans le cadre d'une thèse, de réunir ces trois domaines. Comme l'exprime la figure 39, PUMA nous paraît être le représentant d'une nouvelle famille d'outils, adaptée à la répartition, ainsi qu'aux besoins d'intelligence et d'adaptation (autonomie). C'est ainsi que l'on remarque une remontée en force des langages interprétés, notamment pour la capacité de migration en milieu hétérogène (e.g. Java), certains proposant depuis peu la migration avec conservation de l'activité. De ce point de vue, Agent-Tcl est un exemple frappant, puisqu'il inclut de surcroît la notion d'agent. D'ailleurs, des étudiants de *Dartmouth College* viennent de le mettre en œuvre dans une maquette de Workflow multi-agents ([DARTFLOW]). Avec une dimension beaucoup plus commerciale, le démarche de General Magic est également basée sur des agents implémentés en Telescript, langage orienté objets, interprété, permettant une migration[1] explicite avec conservation de l'activité. L'OMG s'intéresse depuis peu à la notion d'agent ([ORF 96]),

1. Comme pour Java, la transparence à l'hétérogénéité y est assurée par une machine virtuelle.

comme le montre la RFP 3 sur les “common facilities” de CORBA, dont la date de soumission était le 15 avril 96.

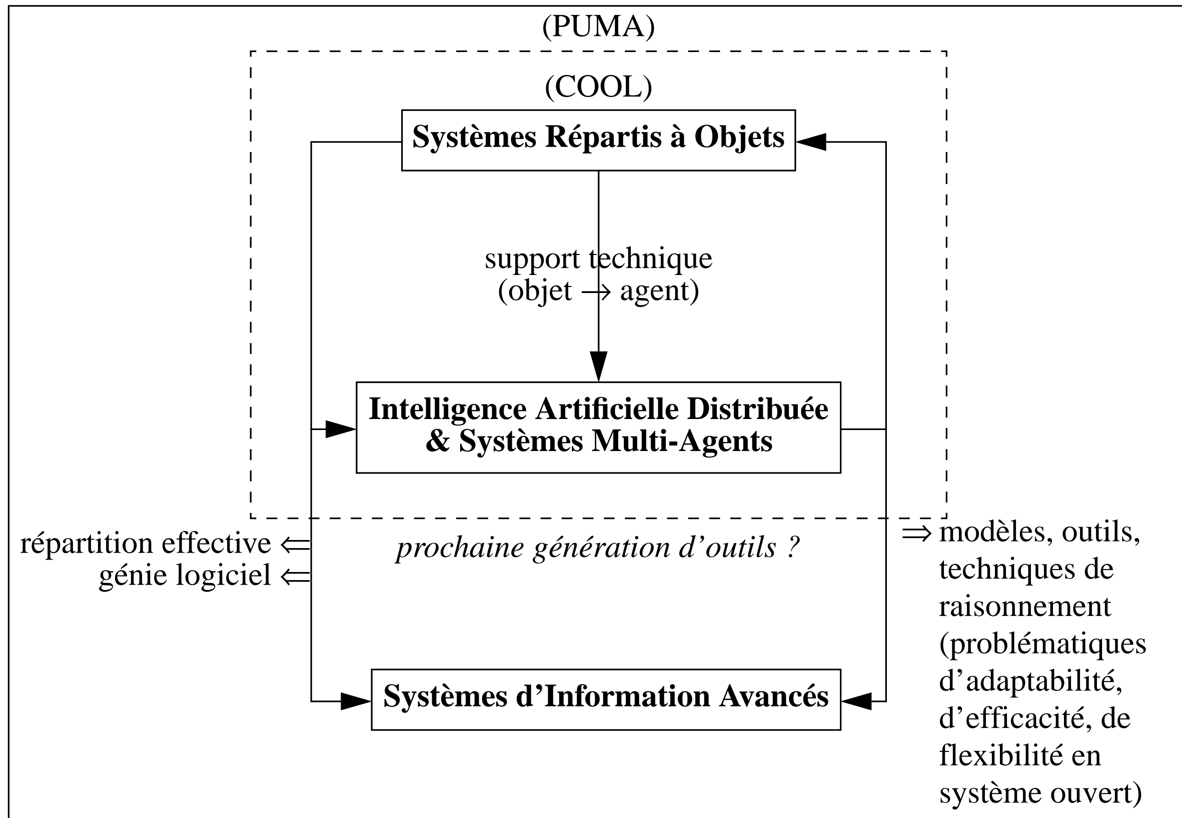


figure 39 : vers une nouvelle génération d’outils pour les systèmes d’information

En ce qui concerne le SEPT, une étude “agents et SRO” est désormais lancée, en partie sur les suites de nos travaux. D’autres systèmes analogues à PUMA ont d’ores-et-déjà été réalisés sur un ORB, avec des langages interprétés aux fonctionnalités plus avancées que le Prolog “classique”. Une thèse se consacre aux structures et protocoles de coopération, notamment au travers d’une généralisation de la notion de groupe[2]. Une seconde thèse s’intéresse à la migration active d’agents[3]. En parallèle, la collaboration avec l’Université de Caen se poursuit pour la recherche de modèles d’agents logiques et de représentation des connaissances adaptés à la bureautique communicante. De plus, la co-réalisation d’un langage interprété basé sur la logique, les contraintes et une orientation objet, permettant un contrôle avancé d’activités, supportant la migration, et destiné à la définition de scripts en environnement CORBA, est en projet. Les autres thèmes de recherche de l’étude “agents” incluent le placement, l’observation et la simulation d’agents sur système réparti à objets. L’étude participe également à

2. En collaboration avec l’Université de Savoie.
3. En collaboration avec le LIB de l’Université de Besançon.

l'encadrement de consultations thématiques lancées par le CNET. Il en résulte des collaborations avec plusieurs laboratoires de recherche :

- l'IRIT (Université de Toulouse) et le CERT (ONERA) développent une approche multi-agents sur le thème de l'*accès intelligent à l'information répartie* ;
- le LIFIA (IMAG Grenoble) s'intéresse *aux architectures d'agents et structures d'interactions pour l'étude de la dynamique des organisations dans les SMA ouverts* ;
- le MASI (Université Paris VI) propose une contribution en matière de *conception, analyse et réalisation intégrale de SMA* (CARISMA).

Sur un plan plus personnel, ces travaux trouvent un prolongement au sein de l'étude "CIDRIA générique"[4] du projet PREVISIA[5], qui permettra d'approfondir et de concrétiser l'approche multi-agents, sur système réparti à objets CORBA, des systèmes de *Workflow.*

4. En collaboration avec l'équipe "agents et SRO" du SEPT.
5. "Plate-forme Répartie d'Evaluation et d'Intégration des Systèmes d'Information Avancés", G.I.E. créé en 1994 par la BNP, l'EDF, le CNET et l'INRIA.

BIBLIOGRAPHIE

Références bibliographiques

[AGH 88] Gul Agha. *Actors, A Model of Concurrent Computation in Distributed Systems.* MIT Press 1988.

[ARC 93] J.P. Archangeli, A. Marcoux, C. Maurel, P. Sallé. *Le langage d'acteur PLASMA II et les systèmes multi-agents.* pp. 181-191, Premières journées francophones IAD & SMA, Toulouse, avril 1993.

[ARO 95] Patrice Aron. *Micronoyaux : la réalité en retard sur les promesses.* Le Monde Informatique, 20 janvier 1995.

[ATK 93] Bill Atkinson, dans Science et Vie Micro numéro 111, p. 61, décembre 1993.

[BAR 93] Michel Barat, Jean Erceau. *Utilité et utilisation d'un principe intégrateur dans un outil de conception de systèmes complexes multi-experts.* Actes du 2ème congrès européen de systémique, vol. III pp 860-869. Prague, octobre 1993.

[BAR 95] Oskar Bartenstein. *Prolog Success Factors — Current Trends in the East.* Keynote speaker, Practical Applications of Prolog, Paris, 1995.

[BAS 92a] Rémi Bastide. *Modélisation de la dynamique des documents : étude comparative de quatre formalismes.* Rapport interne Université Toulouse I - SEPT, 1992.

[BAS 92b] Rémi Bastide. *Objets coopératifs : un formalisme pour la modélisation des systèmes concurrents.* Thèse de doctorat de l'Université Paul Sabatier de Toulouse, 1992.

[BES 88] François Besse. *Etude et implantation d'un algorithme de génération de descriptions formelles du parcours d'un document bureautique.* Rapport de DEA Université de Caen, note technique DT/SPT/SCE/59, SEPT 1988.

[BIO 93] Joëlle Biondi. *Algorithmes génétiques et apprentissage.* 23ème Ecole Internationale d'Informatique de l'AFCET, Neuchâtel, 1993.

[BOU 89] François Bourdon. *CIDRE: intelligent circulation of distributed folders.* 5th International Workshop on Telematics, Denver 1989.

[BOU 93] François Bourdon. *Un modèle de dérive des connaissances - application en bureautique.* Thèse de l'Université du Mans, 1993.

[BOU 94] Danielle Boulanger, Joël Colloc, Gilles Dubois, Caroline Wintergerst. *Objets-Agents : continuum ou différences ?* Journée Systèmes Multi-Agents du PRC (MRT) - GDR (CNRS) Intelligence Artificielle, Paris, décembre 1994.

[BOU 96] François Bourdon. *Computational socialization in global information systems : questions and prospects.* 2nd international conference on Information System Analysis and Synthesis, Orlando, juillet 96.

[BOW 90] John Bowers, Steve Benford. Préface de "Studies in computer supported cooperative work: theory, practice, and design", Elsevier Science Publishers B.V., 1991.

[BYT 94] *Human-Centered Computing.* Byte, pp. 66-67, avril 1994.

[CAM 95] Valérie Camps, Marie-Pierre Gleizes. *Principes et évaluation d'une méthode d'auto-organisation.* pp. 337-348, Journées francophones IAD & SMA, Saint-Baldolph, mars 1995.

[CHA 93] Brahim Chaib-draa, E. Paquet. *Routines, situations familières et non-familières dans les environnements multi-agents.* Rapport interne, département d'Informatique de l'Université Laval, Québec, Canada 1993.

[CHE 93] Vincent Chevrier. *GTMAS : un outil pour la construction et l'évaluation de systèmes multi-agents.* pp. 45-56, Premières journées francophones IAD & SMA, Toulouse, avril 1993

[CORBA2] *The common Object Request Broker: Architecture and Specification (revision 2.0)*, Object Management Group, juillet 1995

[CRCX400] Ambiance CRC/X.400. *Circulation de dossiers sur messagerie.* documentation SEPT, 1994.

[DARTFLOW] Ting Cai, Peter A. Gloor, Saurab Nog. *DartFlow: A Worflow management system on the Web using transportable agents.* Technical Report PCS-TR96-283, Department of Computer Science, Dartmouth College, Hanover NH 03755, mai 1996.

[DES 93] Marc Desreumaux. *Pourquoi les entreprises ont-elles besoin de systèmes d'information flexibles ?* Tome 7, 1er congrès biennal AFCET, Versailles, juin 1993.

[DIL 92] Bruno Dillenseger. *Etude de l'introduction de raisonnement dans la circulation des documents du système CIDRE.* Rapport de stage de DEA Université de Caen - SEPT, 1992.

[DIL 94] Bruno Dillenseger, François Bourdon. *Une approche multi-agents des systèmes de bureautique communicante.* Journée Systèmes Multi-Agents du PRC-IA, Paris, 16 décembre 1994.

[DIL 95a] Bruno Dillenseger, François Bourdon. *Supporting intelligent agents in a distributed environment: a COOL-based approach.* TOOLS 16 pp. 235-246, Prentice Hall, 1995.

[DIL 95b] Bruno Dillenseger, François Bourdon. *Towards a multi-agent model for the office information system: a Prolog-based approach.* Proceedings PAP '95, Paris, pp. 191-200.

[DUP 94] Gérard Dupoirier. *Technologie de la Gestion Electronique de Documents : l'édition électronique.* Hermes, 1994.

[ELL 79] Clarence A. Ellis. *Information Control Nets: A mathematical model of office information flow.* Conference on Simulation, Measurement and Modeling of Computer Systems - 1979.

[ERC 93] Jean Erceau. *Intelligence Artificielle Distribuée et Systèmes Multi-Agents — de la théorie aux applications.* 23ème Ecole Internationale d'Informatique de l'AFCET, Neuchâtel, 1993.

[FER 92] J. Ferber, B. Habert, F.X. Testard-Vaillant, P. Estraillier. *MARSALA — Modélisation, analyse et réalisation de systèmes d'agent par langages d'acteurs.* Journée IBP, septembre 1992.

[FER 93] Jacques Ferber. *Introduction à l'intelligence artificielle distribuée et aux systèmes multi-agents.* 23ème Ecole Internationale d'Informatique de l'AFCET, Neuchâtel, 1993.

[FER 95] Jacques Ferber. *Les systèmes multi-agents.* InterEditions, 1995.

[FRE 93] Dominique Fresneau. *Analyse et modélisation des comportements sociaux dans des univers multi-agents vivants.* 23ème Ecole Internationale d'Informatique de l'AFCET, Neuchâtel, 1993.

[GAS 91] Les Gasser. *Social conceptions of knowledge and action: DAI foundations and open systems semantics.* Artificial Intelligence 47, pp. 107-138, Elsevier 1991.

[GID 94] John Gidman. *Practical applications of distributed object technology.* Object Magazine, mars/avril 94, pp. 40-43.

[HAB 89] Sabine Habert. *Gestion d'objets et migration dans les systèmes répartis.* Thèse de Doctorat de l'Université Paris-VI, décembre 1989.

[HAS 92] Salima Hassas. *GMAL — Un modèle d'acteurs réflexif pour la conception de systèmes d'intelligence artificielle distribuée.* Thèse de doctorat, Université Claude Bernard - Lyon I, 1992.

[HAU 93] Hans Haugeneder, Donald Steiner. *IMAGINE's goals and approach.* IMAGINE Final project report, ESPRIT project 5362.

[HEW 77] C. Hewitt. *Viewing control structures as patterns of passing messages.* Artificial Intelligence, vol. 8, pp. 323-364, 1977.

[HEW 91] Carl Hewitt. *Open Information Systems Semantics for DAI.* Artificial Intelligence 47, pp. 79-106, Elsevier 1991.

[HØI 90] Steinar Høistad. *Entrevue avec Steinar Høistad, directeur général Europe, Unix International*. ST Magazine No 39, avril 1990, pp. 212-218.

[HOV 95] David Hovel. *Using Prolog in Windows NT network configuration.* The third international conference on the Practical Application of Prolog, pp. 317-339, avril 1995.

[HUH 87] M. Huhns. *Distributed Artificial Intelligence.* London, Pitman, 1987.

[HYM 95] Emmanuel Hym. *Intégration et mise en œuvre d'un langage interprété sur la plate-forme répartie à objet COOL v2.* Rapport de stage DEA Université de Caen - SEPT, septembre 1995.

[JAM 90] Ang. JAMES. *A comprehensive office modeling framework for ITHACA.* ITHACA.BULL.89.D9.#1, BULL S.A. 1989.

[KAR 90] Bernhard H. Karbe, Norbert G. Ramsperger. *Influence of Exception Handling on the Support of Cooperative Office Work.* Conference on Multi-User Interfaces and Applications, projet ProMInanD 1990.

[LEA 93] Rodger Lea, Christian Jacquemot, Eric Pillevesse. *COOL: system support for distributed programming.* Communications of the ACM, Vol.36, No.9, 1993.

[LEF 95] Claire Lefèvre. *Agents Logiques Communicants.* Thèse de doctorat de l'Université de Caen, 13 février 1995.

[LEM 82] Pierre Lemaître, J.F. Begouen Demeaux. *Pratique d'organisation des services administratifs.* Les Editions d'Organisation, 1982.

[LIN 90] Rob van der Linden, Joe Sventek. *ISA Project — trading in the five projections.* rapport "Advanced Networked Systems Architecture" APM/RC.101.02, Architecture Projects Management Limited, 1990.

[MAL 95] Eric Malville. *Etude d'une architecture agent sur la plate-forme chorus/COOL v2.* Rapport de stage DEA Université de Caen - SEPT, septembre 1995.

[MAL 96] Eric Malville, Anne Lille, François Bourdon. *Des outils de développement de Systèmes Multi-Agents sur COOL v2.* Contrôle Réparti dans les Applications Coopératives. Paris, mai 96.

[MER 94] François Merciol. *Hétérogénéité dans les systèmes répartis à objets.* Thèse de doctorat de l'université Paris-VI, décembre 1994.

[MET 89] Denis Metral-Charvet, François Saint-Lu. *Le système Chorus : Temps Réel, Répartition et UNIX intégrés.* Document CS/TR-89-3.2, Chorus Systèmes, 1989.

[MOU 93] Bernard Moulin, Louis Cloutier. *Collaborative work based on multiagent architectures: a methodological perspective.* "Artificial Intelligence: theory and application", Prentice Hall 1993.

[NEW 94] David S. Newman. *Strategic planning for distributed object management.* Object Magazine, juin 94, pp. 74-77.

[ODP 95] *Reference Model of Open Distributed Processing.* Norme ISO 10746 et recommandations ITU-T X.900, janvier 95.

[ORF 96] Robert Orfali, Dan Harkey, Jeri Edwards. *The essential distributed objects survival guide.* Wiley, 1996.

[PIL 89] Eric Pillevesse. *Techniques “objet” pour la représentation des droits d’accès sur des parties de document et le traitement des visas dans un document.* Woodman89/CET/SEPT/SCE/ARC.

[POP 93] Philippe Populaire, Olivier Boissier, Jaime Suchman. *Description et implémentation de protocoles de communication en univers multi-agents.* Premières journées francophones IAD & SMA, Toulouse 1993.

[PREVISIA] Plaquette de présentation du projet PREVISIA, GIE PREVISIA c/o INRIA Rocquencourt, 10 novembre 1995.

[PUJ 95] Guy Pujolle. *Les réseaux.* pp. 674-684, Eyrolles 1995.

[RAS 94] Daniel W. Rasmus. *Incorporating rules with objects — A hybrid methodology for decision support.* Object Magazine juin 1994, pp. 27-46.

[RCOX400] Ambiance RCO/X.400. *Rédaction coopérative de documents sur messagerie.* documentation SEPT, 1994.

[ROY 94] Mark Roy, Alan Ewald. *Locating and managing ORB objects.* Object Magazine, mars/avril 94, pp. 28-32.

[SCH 92] Eric Schwartz. *Un modèle générique de l’émergence et de l’évolution des systèmes naturels.* Deuxième Ecole AFCET de Systémique, Mont Ste Odile, octobre 1992.

[SHO 93] Yoav Shoham. *Agent-oriented programming.* Artificial Intelligence 60 (1993), Elsevier, pp. 51-92.

[SMI 80] Reid G. Smith. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.* IEEE Transactions on Computers, vol. c-29, No 12, december 1980.

[STE 86] Jean-Bernard Stefani. *Bureautique : un tour d’horizon.* Document Technique DT/SPT/SCE/2, SEPT, juillet 1986.

[STI 93] Serge Stinkwich. *Modèle et environnement objet dédié aux systèmes multi-agents.* pp. 205-216, Premières journées francophones IAD & SMA, Toulouse, avril 1993.

[STI 94] Jim Stikeleather. *Why distributed object computing is inevitable.* Object Magazine, mars/avril 94, pp. 35-39.

[TAN 92] Andrew Tanenbaum. *Réseaux — Architectures, protocoles, applications.* InterEditions 1992.

[THI 93] R.A. Thiétart. *Ordre et Chaos dans les Organisations.* Journée d'étude AFCET "Les Systèmes d'Information, Autonomie et Chaos", Paris, 24 novembre 1993.

[TRO 92] Christophe Trompette. *Etude de modèles de négociation dans un univers multi-agent.* Rapport de stage de DEA Université de Caen - SEPT, 1992.

[VAN 94] Gérard Vandome. *Présentation OMG.* Bull/OSS/MIO/DOM, 1992.

[VER 90] J.P. Verjus. *Introduction aux systèmes informatiques répartis.* Ecole d'été "Construction des Systèmes d'Exploitation Répartis" INRIA 1990.

[WHI 94] J. White. *Telescript technology: the foundation for the electronic market place — General Magic White Paper*, General Magic, 1994.

[WIE 94] Jan Wielemaker. *SWI-Prolog 1.8 Reference Manual.* University of Amsterdam, janvier 94.

ANNEXES

Annexe A : documentation PUMA

Sommaire

I- langage Prolog étendu

a) généralités

b) restrictions

c) les prédicats PUMA

d) données “réservées”

II- l’utilisation des classes du kit PUMA

a) arbre d’héritage

b) la classe ‘puma’

c) les classes ‘bureau’, ‘interp’ et ‘agent’

d) la classe objetSMA

I- langage Prolog étendu

a) généralités

Le dialecte de PUMA est C-Prolog, avec certaines restrictions, complété par des prédicats spécifiques -dits “prédicats PUMA”- qui reprennent pour la plupart des fonctionnalités de COOL.

b) restrictions

1- concernant les prédicats PUMA :

- ils ne réussissent qu'une fois (cf. backtrack) ;
- lors du chargement d'un fichier Prolog contenant certains types d'erreur, l'objet PUMA plante.

2- concernant le simulateur Chorus

Pour permettre la migration entre deux stations, il faut qu'elles partagent le même répertoire "/tmp" (montage NFS).

3- divers

- la seule façon de migrer un objet PUMA est l'auto-migration de celui-ci par appel au prédicat migrate/1 (ou follow/0 dans le cas de l'utilisation de la classe `objetSMA`) ;
- la communication synchrone entre deux objets (i.e. accès direct à la base de connaissance d'un autre objet) n'est possible que s'ils partagent le même contexte COOL.

c) les prédicats PUMA

1- communication

- myname(?N) unifie N avec le nom de l'objet PUMA, ou échoue s'il n'est pas enregistré dans le service de nommage COOL.
- vanish retire l'objet PUMA du service de nommage. Echoue si l'objet n'est pas enregistré dans le serveur de nommage du site où il se trouve.
- appear(+N) enregistre l'objet PUMA sous le nom N auprès du service de nommage COOL. Echoue s'il est déjà nommé.
- available(+N) réussit si le nom N est présent dans le service de nommage COOL, échoue sinon.
- send(+N, +M) envoie le terme M comme message à l'objet de nom N. Echoue si le destinataire n'existe pas ou si le message ne lui parvient pas.
- receive(?M) unifie M avec le premier message non lu de la boîte aux lettres (FIFO). Echoue s'il n'y a pas de message en attente.
- addtogroup(+G) ajoute l'objet dans le groupe de communication nommé G. Le groupe est créé s'il n'existe pas. Echoue en cas de problème COOL d'ajout ou de création du groupe.
- mygroups(?L) unifie L avec la liste des groupes de communication auxquels l'objet appartient.

- quitgroup(+G) retire l'objet du groupe de communication de nom G. Echoue en cas de problème de retrait, réussit sinon.
- broadcast(+G, +M) diffuse le terme M en message aux membres du groupe de nom G. Echoue si le groupe n'existe pas.
- functmode(+G,+M) envoie le terme M en message à un membre quelconque du groupe G. Echoue si le groupe n'existe pas.
- migrate(+N), afin de migrer proprement vers un autre objet COOL, réalise l'ensemble des opérations suivantes ou échoue si le destinataire n'existe pas :

(1) retrait du serveur de nommage du site courant ;

(2) rupture d'une éventuelle liaison synchrone établie, par appel à hangup/0 ;

(3) migration vers l'objet de nom N, accompagnée du message 'puma(object(x))' avec 'x' nom de l'objet PUMA migrant (ou 'puma(object)' si l'objet n'a pas de nom) ;

(4) enregistrement auprès du service de nommage du site destination.

NB : Pour des raisons spécifiques à COOL, la migration n'est effective que lorsque l'objet destinataire lit le message d'accompagnement. Par ailleurs, les éventuelles communications synchrones acceptées ne sont plus viables (il faut donc s'en préoccuper avant de migrer).

- ringup(+N,+D) envoie un message de requête de communication synchrone à l'objet de nom N. Le message envoyé est du type 'puma(ringup(r))' ou 'r' est le nom de l'objet qui réclame la connexion. Le prédicat reste bloqué en attente d'une réponse pendant un délai de D millisecondes. Echoue s'il n'existe pas d'objet de nom N, ou si aucune réponse n'est reçue, ou si la reponse reçue est défavorable.
- ringup(+N) est une version de ringup/2 avec délai d'attente infini.
- accept accepte la requête de communication synchrone contenue dans le dernier message lu. Un tel message est de la forme 'puma(ringup(r))' ou 'r' est le nom de l'objet demandeur. Echoue si aucune requête ne vient d'être lue.

  NB : il ne faut faire aucun appel à receive/1 entre la réception du message 'puma(ringup(r))' et la réponse accept/0.
- refuse refuse la requête de communication synchrone contenue dans le dernier message lu. Un tel message est de la forme 'puma(ringup(r))' ou 'r' est le nom de l'objet demandeur. Echoue si aucune requête ne vient d'être lue.

  NB : il ne faut faire aucun appel à receive/1 entre la réception du message 'puma(ringup(r))' et la réponse refuse/0.

- phone(+B) fait dériver le but B par l'objet PUMA ayant accepté l'établissement d'une communication synchrone (accès direct à son noyau Prolog). Echoue s'il n'y a pas de communication synchrone établie.
- hangon(?N) unifie N avec le nom de l'objet ayant accepté l'établissement d'une communication synchrone (accès direct à son noyau Prolog). Echoue s'il n'y a pas de communication synchrone établie.
- hangup met fin à la communication synchrone établie et en informe l'objet concerné par un message de la forme 'puma(hangup(x))' ou 'x' est le nom de l'objet qui rompt la communication. Echoue s'il n'y avait pas de communication synchrone établie.
- follow provoque la migration (cf. migrate/1) vers l'objet ayant envoyé le dernier message lu, à condition que ce message ait été envoyé par la primitive COOL `objectCall()`[1]. Echoue si la migration n'a pas eu lieu.

2- divers

- load(+F) charge la base Prolog contenue dans le fichier F. Si F est l'atome 'user', c'est l'entrée standard qui est consultée. Sinon, le fichier est cherché dans le répertoire $PUMA_DIR/bin/prolog, puis dans le répertoire courant. Echoue si le fichier n'existe pas.
- create(+C, +N, +F) crée un objet COOL de la classe C dans le contexte de l'objet PUMA appelant, les arguments N et F étant passés au constructeur de la classe C sous forme de chaînes de caractères[2]. Réussit si l'objet est crée, échoue en cas de problème COOL.
- interrupt(+S) interrompt le programme Prolog en cours et redonne la main au niveau C++. Toute interruption est traitée par défaut en affichant S et en attendant une entrée au clavier. Echoue si le caractère "fin de fichier" est rencontré, réussit sinon. La chaîne lue au clavier pourra être récupérée par ireturn/1[3].
- ireturn(?R) unifie R avec le résultat rendu par le niveau C++, suite à la dernière invocation de interrupt/1. Echoue si aucune interruption n'a eu lieu ou si la dernière interruption n'a fourni aucune valeur de retour.

---

1. Ce prédicat permet à un objet PUMA de suivre un objet COOL héritant de la classe objetSMA auquel il est lié et qui aurait migré.
2. L'utilisation typique consiste à créer un objet PUMA (e.g. de la classe 'interp' ou 'agent') de nom 'N', avec le fichier 'F' comme base Prolog initiale.
3. Ce comportement peut être substitué en surchargeant la méthode 'puma::interrupt_handler()'.

d) données “réservées”

1- Base de données “système”

Un certain nombre de données “système” concernant le fonctionnement de PUMA sont stockées dans une base Prolog sous la clé ‘puma’. Par conséquent, il ne faut pas utiliser les prédicats recorda, recordz et recorded avec ‘puma’ comme clé.

2- Messages “système”

Certains messages sont engendrés par PUMA lui-même, sans que le prédicat send/2 soit invoqué. Ces messages sont du type ‘puma(Terme_Prolog)’. Un programme Prolog-PUMA ne doit donc jamais envoyer explicitement de message de cette forme. En revanche, il doit tenir compte de ceux qu’il reçoit :

- puma(ringup(n)) (cf. ringup/1, ringup/2, accept/0, refuse/0) ;
- puma(hangup(n)) (cf. hangup/0) ;
- puma(object) ou puma(object(n)) (cf. migrate/1, follow/0).

De plus, lorsqu’un agent est lié a un objet COOL par le biais de l’utilisation de la classe `objetSMA`, il est susceptible de recevoir des messages du type ‘objetSMA(Terme)’. La version actuelle de `objetSMA` engendre uniquement des messages ‘objetSMA(follow)’ (cf. prédicat follow/0 et classe `objetSMA`).

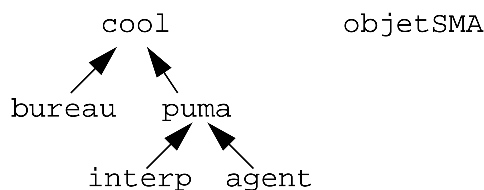
3- Procédures “réflexes”

Un certain nombre de réflexes peuvent être définis dans un programme Prolog-PUMA :

- puma/0, réflexe de création de la classe `puma` ;
- objetSMA/1, réflexes de la classe `objetSMA`.

II- Utilisation des classes du kit PUMA

a) arbre d’héritage



- Les objets de la classe `bureau` sont des objets serveurs actifs qui scrutent en permanence leur porte de communication, affichent à l’écran les messages reçus et accueillent les objets migrants.

- La classe `puma` est celle qui intégre un noyau Prolog et dispose de l'interface complète d'accès et de contrôle pour ce noyau. Elle possède une porte de communication mais pas d'activité.
- La classe `interp` hérite de la classe `puma` et définit une activité qui consiste en un interpréteur interactif de PUMA.
- La classe `agent` hérite de la classe `puma` et définit une activité générique d'agent par une boucle d'appels au but prolog activite/0 (à programmer dans un fichier Prolog).
- La classe `objetSMA` est une classe virtuelle dont le rôle est d'associer un objet `agent` à un objet `cool`.

b) interface de la classe 'puma'

1- notion d'état Prolog

Le noyau Prolog est caractérisé par un état qui peut prendre principalement les valeurs :

PYES	Le dernier but (sans variable) a réussi.
POK	La dernier but a réussi et le résultat de l'unification est disponible.
PFAIL	Le dernier but a échoué.
PERROR	Le dernier but était incorrect du point de vue Prolog.
PEND	La session Prolog est terminée.

2- le constructeur

`puma(char *s1, char *s2)`

- `s1` est le nom symbolique sous lequel l'objet sera enregistré dans le service de nommage COOL ;
- `s2` est le nom du fichier Prolog à charger pour initialiser la base du noyau (NB : la non-existence de ce fichier ne perturbe pas la création de l'objet). Dès que la base est chargée, le but puma/0 est invoqué. Ceci permet de définir un réflexe de création dans le programme Prolog.

3- les méthodes publiques

`int puma_P()`

Acquisition du sémaphore d'accès au noyau Prolog. Retourne 0 en cas d'exécution correcte.

---

int puma_P(int d)

Acquisition en un temps fini du sémaphore d'accès au noyau Prolog. Retourne 0 en cas de succès. Retourne la constante Chorus K_ETIMEOUT si le délai de `d` millisecondes est expiré.

---

int puma_V()

Libération du sémaphore d'accès au noyau Prolog. Retourne 0 en cas d'exécution correcte.

---

int puma_call(char *s1, char *s2=0)

Appelle la requête prolog s1 et retourne le nouvel état Prolog.

- Si l'état Prolog retourné est POK, le résultat de l'unification est copié dans `s2` (si ce paramètre est spécifié) suivant le format "Var = val : "...
- Si l'état Prolog retourné est PINTERRUPT, l'argument passé par l'appel au prédicat interrupt/1 est copié dans `s2` (si ce paramètre est spécifié).
- Si l'état Prolog retourné est PERROR, le message d'erreur est copié dans `s2` (si ce paramètre est spécifié).

---

int puma_recall(char *s1=0, char *s2=0)

Si l'état Prolog courant est POK, force un échec pour tenter une autre unification (backtrack), ne fait rien sinon. Retourne le nouvel état Prolog.

- Si l'état Prolog retourné est POK, le résultat de l'unification est copié dans `s2` (si ce paramètre est spécifié) suivant le format "Var = val : "...
- Si l'état Prolog retourné est PINTERRUPT, l'argument passé par l'appel au prédicat interrupt/1 est copié dans `s2` (si ce paramètre est spécifié).
- Si l'état Prolog retourné est PERROR, le message d'erreur est copié dans `s2` (si ce paramètre est spécifié).

---

int puma_unif(char *s)

Copie le résultat de la dernière unification Prolog dans la chaîne de caractères `s` suivant le format "Var = val : "... Retourne 0 si l'état Prolog courant n'est pas POK, 1 sinon (`s` contient alors le résultat de l'unification).

---

int puma_unif(char *s1, char *s2)

Suite à une unification Prolog, copie la valeur de la variable Prolog de nom `s1` dans `s2`. Retourne 0 si Prolog n'est pas dans l'état POK ou si `s1` ne figure pas parmi les variables de l'unification, retourne 1 sinon (`s2` contient alors la valeur de la variable de nom `s1`).

---

void puma_interp()

Interpréteur interactif Prolog/PUMA. La fonction retourne en cas de fin de session Prolog (état PEND).

---

void puma_init()

Initialisation de PUMA à la création ou après migration :

- initialisation du contexte Prolog,
- enregistrement auprès du service de nommage local,
- chargement du fichier-état Prolog ("startup" ou fichier de migration),
- chargement éventuel d'une base Prolog initiale (création uniquement) et appel au prédicat puma/0 afin de démarrer une activité Prolog si ce prédicat est défini dans la base chargée,
- restauration de la boîte aux lettres (après migration uniquement),
- déblocage du sémaphore d'accès au noyau Prolog.

---

void puma_exit()

Termine la session Prolog — en particulier :

- retire l'objet du service de nommage local (s'il y a lieu),
- retire l'objet des groupes auxquels il appartient (s'il y a lieu),
- avertit le noyau Prolog (=> l'état Prolog devient PEND).

4- gestionnaire d'interruption

Le prédicat interrupt/1 est principalement destiné à permettre au programmeur d'invoquer des fonctions C/C++/COOL depuis un programme Prolog. Lorsque ce prédicat est appelé, il déclenche un appel à la méthode `interrupt_handler(char* in, char* out)` qui définit un comportement par défaut en réponse à toute interruption :

- affichage de l'argument de interrupt/1,
- lecture au clavier,

- échec si le caractère de fin de fichier est rencontré,
- succès sinon et mémorisation du terme lu pour un éventuel appel ultérieur au prédicat ireturn/1.

Pour pouvoir redéfinir cette méthode (e.g. par héritage), il suffit de savoir que :

(1) `in` est la chaîne de caractères contenant l'argument de interrupt/1 ;

(2) `out` doit contenir le terme Prolog de retour, destiné à être unifié avec l'argument de ireturn/1 lors d'un éventuel appel ultérieur à ce prédicat ;

(3) la méthode doit retourner 0 en cas d'échec, ou un entier non nul sinon.

5- quelques précisions

installation du noyau Prolog. Le noyau Prolog doit être installé à la création ou après migration par la méthode `puma_init()` avant toute invocation. Avant de quitter, il est vivement conseillé de faire appel à la méthode `puma_exit()`.

exclusion mutuelle. Les fonctionnalités de COOL permettent à plusieurs objets d'invoquer des méthodes sur un même objet. Or le noyau Prolog doit être invoqué en exclusion mutuelle. Ainsi, avant d'invoquer les méthodes `puma_call()` et `puma_interp()`, il faut acquérir le sémaphore par appel à `puma_P()`. Ensuite, on doit garder le contrôle du noyau pour toute invocation `puma_recall()` ou `puma_unif()`. Enfin, on libère le sémaphore par appel à la méthode `puma_V()`.

niveau C++ / niveau Prolog. Si la méthode `puma_call()` permet au niveau C++ d'invoquer le niveau Prolog, réciproquement, le prédicat interrupt/1 permet d'invoquer le niveau C++ depuis Prolog et de récupérer un paramètre de retour via le prédicat ireturn/1.

c) utilisation des classes dérivées de COOL

1- classe 'bureau'

Le constructeur prend une chaîne de caractères en argument. Son contenu est le nom symbolique sous lequel l'objet sera enregistré auprès du service de nommage COOL. Un objet `bureau` affiche tous les messages qu'il reçoit et permet principalement d'accueillir tout objet `cool` désirant migrer dans son contexte.

2- classe 'interp'

Le constructeur est celui de la classe `puma`. Dès que la base initiale est chargée et l'éventuel prédicat puma/0 invoqué, on se retrouve sous l'interpréteur interactif. Lorsque

qu'une dérivation de but rencontre le prédicat interrupt/1, l'argument est affiché et l'interpréteur demande la valeur de retour souhaitée par l'utilisateur (cf. prédicat ireturn/1).

On peut sortir de l'interpréteur par les prédicats C-prolog end/0, halt/0, quit/0, end_of_file/0 ou en entrant le caractère "fin de fichier" (généralement CTRL-D).

L'exécution de cet objet interpréteur se fait entièrement en exclusion mutuelle, ce qui fait qu'il est impossible d'accéder à son noyau Prolog depuis un autre objet COOL. En particulier, il est impossible d'établir une communication synchrone avec lui (cf. prédicat phone/1). Ainsi, il devra systématiquement invoquer refuse/0 pour tout message "ringup" (cf. prédicats ringup/1/2) reçu.

3- classe 'agent'

Le constructeur est celui de la classe `puma`. Dès que la base initiale est chargée et l'éventuel prédicat puma/0 exécuté, l'objet entre dans une boucle d'appels au prédicat activite/0. Chaque appel est précédé par une réquisition et suivi d'une libération du sémaphore d'accès, afin de permettre des invocations synchrones sur le noyau Prolog de l'objet.

L'activité de l'objet prend fin lorsque l'état Prolog devient PEND. L'objet fait alors appel à la méthode `puma_exit()` avant de terminer.

d) la classe objetSMA

1- Généralités

Toute classe dérivant de COOL et qui hérite de la classe `objetSMA` s'associe à un objet de la classe `agent`. Ainsi, par appel au constructeur de `objetSMA`, on crée un objet `agent` dont on conserve l'adresse et la capacité de communication afin de pouvoir l'invoquer ultérieurement de façon synchrone ou asynchrone :

- l'attribut `agentAddr` contient l'adresse de l'objet `agent`. Exemple :

```
agentAddr->puma_P();
agentAddr->puma_call("but_prolog(X).");
agentAddr->puma_unif("X", str);
agentAddr->puma_V();
```

NB : ne jamais oublier l'exclusion mutuelle pour toute invocation synchrone du noyau prolog.

- l'attribut `agentMbox` contient la capacité de communication de l'agent, i.e. l'adresse de sa boîte aux lettres. Exemple :

```
objectSend(&agentMbox, &msg);
```

NB :

(1) Ne pas utiliser `objectCall()` ! (en dehors des cas particuliers follow/0, refuse/0 et accept/0, il n'existe pas de prédicat PUMA prévu pour faire un appel `objectReply()`).

(2) Ne pas oublier de faire un appel `agentInstall(this)` pour installer l'agent à la création et après migration (l'agent suit systématiquement l'objet auquel il est lié).

2- Constructeur

`objetSMA(char *s1, char *s2)`

Le constructeur crée un objet de la classe `agent` et conserve son adresse ainsi que sa capacité de communication dans les attributs `agentAddr` et `agentMbox`. Les paramètres `s1` et `s2` sont transmis au constructeur de la classe `agent` :

- `s1` est le nom symbolique le l'agent créé (service de nommage COOL) ;
- `s2` est le nom du fichier contenant la base Prolog initiale à charger.

3- Installation, destruction

`void agentInstall(cool* p)`

Installe l'agent après sa création ou après migration. `p` doit être l'adresse de l'objet auquel l'agent est lié. Par conséquent, l'appel se fait toujours au début de la méthode `main()` de l'objet par : "`agentInstall(this);`".

Lorsque l'appel fait suite à une migration, le message objetSMA(follow) est envoyé à l'agent qui doit ensuite suivre l'objet auquel il est lié via le prédicat follow/0.

Par ailleurs, cet appel déclenche un réflexe, s'il est défini dans la base Prolog de l'agent :

- objetSMA(init) en cas de création ;
- objetSMA(reinit) en cas de réinstallation après migration.

NB : Le réflexe objetSMA(init) est exécuté postérieurement au réflexe de création puma/0. La différence entre ces deux réflexes réside dans le fait que puma/0 est exécuté dans une

activité parallèle (celle de l'agent) alors que les réflexes objetSMA/1 sont exécutés par l'activité de l'objet dérivant de la classe `objetSMA`.

`void agentKill()`

Provoque la disparition "propre" de l'agent après avoir tenté d'invoquer le réflexe objetSMA(exit) de l'agent.

4- Méthode utilitaire

`int list2string(char *l, char *s=NULL)`

Traduit la chaîne `l` au format Prolog (liste Prolog d'entiers) en une chaîne `s` au format C. Si `s` vaut NULL ou n'est pas spécifié, la traduction est copiée dans `l`. Retourne 0 en cas d'erreur.

Annexe B : “workflow” multi-agents

Les agents *document* doivent exécuter un enchaînement particulier d’actions (workflow décrit par un “Information Control Net”, figure 40) en invoquant des agents *imprésario*, représentant les utilisateurs. Ces actions, qui correspondent typiquement à la valuation d’un champ du document, sont déclenchées de manière asynchrone et parallèle[1] par l’intermédiaire de messages. Chaque impresario invoque en conséquence l’utilisateur qu’il représente, et retourne le résultat (typiquement une valeur de champ). Chaque agent document gère un historique des requêtes envoyées et des réponses reçues. Bien entendu, plusieurs documents peuvent s’exécuter simultanément.

La procédure, purement fictive, s’inspire d’un bon de commande. Elle comprend un nombre variable d’actions suivant les valeurs des champs. Ainsi, le nombre de visas nécessaires augmente avec la valeur du champ “prix”.

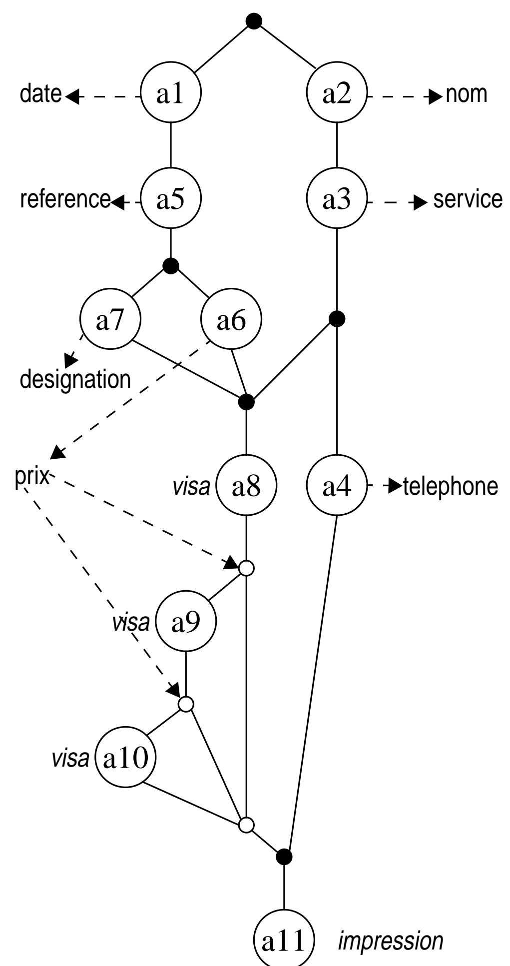


figure 40 : exemple fictif d’ICN

1. suivant les spécifications de l’ICN.

Agent impresario

Prédicats d’utilité générale

% **ajoute**(X,L,R)

R est la liste obtenue en ajoutant X (si X est un atome) ou les elements de X (si X est une liste, et de maniere recursive) en tete de la liste L; les atomes figurant deja dans L ne sont pas ajoutes.

```
ajoute([],L,L) :-
    !.
ajoute([X|Q],L,R) :-
    !,
ajoute(X,L,L1),
ajoute(Q,L1,R).
ajoute(X,L,L) :-
    member(X,L),
    !.
ajoute(X,L,[X|L]).
```

% **supprime**(X,L,R)

R est la liste L debarrassee de toutes les occurrences de X

```
supprime(X,[],[]).
supprime(X,[X|Q],R) :-
    !,
supprime(X,Q,R).
supprime(X,[Y|L],[Y|R]) :-
    supprime(X,L,R).
```

% **retire**(X,L1,L2)

L2 est la liste L1 moins la premiere occurrence de X

```
retire(X,[X|L],L) :-
    !.
retire(X,[Y|L1],[Y|L2]) :-
    retire(X,L1,L2).
```

% **inclus**(L1,L2)

la liste L2 contient les elements de la liste L1

```
inclus([],[]).
inclus([],[_|_]).
inclus([X|L1],L2) :-
    retire(X,L2,L3),
inclus(L1,L3).
```

% **try**(B)

tente le but B et reussit en cas d’echec de B

```
try(B) :-
    call(B),
    !.
try(_).
```

```
% append(L1,L2,R)
    R est la concatenation des listes L1 et L2
append([],L,L).
append([X|L1],L2,[X|R]) :-
    append(L1,L2,R).

% member(X,L)
    X appartient a la liste L
member(X,[X|L]).
member(X,[_|L]) :-
    member(X,L).

% listedes(X,P,R)
    R est la liste des X verifiant P
listedes(X,P,R) :-
    bagof(X,P,R),
    !.
listedes(X,P,[]).
```

activité de l’agent

```
puma :- activite.
% NB l'activite est lancee automatiquement par le reflexe de creation PUMA

activite :-
    myname(X),
    write('je suis l''impresario de '),
    write(X),
    nl,
    repeat,
        try((receive(X),traite(X))), % traiter un message
        tache, % tache propre
        fail.

% tache propre de l'agent impresario : rien de particulier
tache.
```

traitement des messages

```
% affichage preliminaire de tous les messages recus (=>traces d'execution)
traite(X) :-
    write('message recu : '),
    write(X),
    nl,
    fail.

% ordre de destruction de l'agent
traite(suicide) :-
    halt.

% requete demandant le remplissage d'un champ
```

```
traite([Source,[Id,remplir(C)]]) :-
    !,
    write(Source),
    write(' demande de remplir le champ '),
    write(C),
    write(' : '),
    read(V), % l'utilisateur donne la valeur du champ
    send(Source,termine(Id,V)), % et l'agent envoie le resultat au demandeur
    write('reponse envoyee'),
    nl.

% requete demandant tout autre action
traite([Source,[Id,Action]]) :-
    write(Source),
    write(' reclame '),
    write(Action),
    write('; taper Enter si Ok >'),
    get(_), % l'utilisateur signale que l'action est accomplie
    send(Source,termine(Id,_)), % notification du demandeur par message
    write('message envoye'),
    nl.
```

Agent document

Prédicats d'utilité générale

idem agent impresario

Prédicats utilitaires concernant les ICN

prédicats généraux

```
% noeud(X)
    X est un noeud (action ou connecteur) de l'ICN.
noeud(X) :-
    action(X,_,_).
noeud(X) :-
    connecteur(X,_).

% debut(X)
    X est le premier noeud de l'ICN
debut(X) :-
    noeud(X),
    not precede(_,X).

% franchi(A,N)
    les noeuds de la liste N ont ete franchis (ou sont franchissables s'il s'agit de connecteurs logiques), A etant la
    liste des actions terminees.
franchi(_,[]) :-
    !.
```

```
franchi(A,[X|L]) :-
    !,
    franchi(A,X),
    franchi(A,L).
franchi(A,X) :-
    member(X,A).
franchi(A,X) :-
    connecteur(X,et),
    !,
    listedes(Y,precede(Y,X),L),
    franchi(A,L).
franchi(A,X) :-
    connecteur(X,_),
    !,
    precede(Y,X),
    franchi(A,Y).
```

% **passant**(P,X)

franchir le noeud X ne conduit pas a un chemin bloquant, sachant que P est la listes des actions declenchees (terminees ou non)

```
passant(P,X) :-
    chemin(C),
    inclus([X|P],C).
```

% **suite**(P,A,X,L)

L est la liste des actions declenchables suite a la terminaison l'action X, sachant que P est la liste des actions declenchees (terminees ou non) et que A est la liste des actions terminees (qui contient deja X). Si l'action X n'a pas de noeud successeur dans l'ICN, L est unifie avec l'atome 'termine' (cela signifie que le Workflow est termine, car l'ICN est un treillis).

```
suite(P,A,X,L) :-
    precede(X,Y),
    !,
    supprconn(P,A,Y,L).
suite(P,A,X,termine).
```

% *remplacer recursivement les connecteurs logiques par les actions possibles par le biais des precedences.*

```
supprconn(_,_,[],[]) :-
    !.
supprconn(P,A,[X|R],L) :-
    !,
    supprconn(P,A,X,L1),
    supprconn(P,A,R,L2),
    append(L1,L2,L).
supprconn(P,A,X,[X]) :-
    action(X,_,_),
    passant(P,X).
supprconn(P,A,X,L) :-
    connecteur(X,et),
    franchi(A,X),
```

```
    !,
    listedes(Y,precede(X,Y),L1),
    supprconn(P,A,L1,L).
supprconn(P,A,X,[]) :-
    connecteur(X,et).
supprconn(P,A,X,L) :-
    connecteur(X,test),
    test(X,Y),
    supprconn(P,A,Y,L).
supprconn(P,A,X,L) :-
    connecteur(X,ou),
    precede(X,Y),
    supprconn(P,A,Y,L).
```

analyse de l’ICN

recherche des parcours possibles (parcours = liste des noeuds traversés) et de la liste des actions (non ordonnée) pour chacun d’entre eux.

% **analyse**
assertion des listes d’actions (non ordonnee) des parcours possibles

```
analyse :-
    analyse(C),
    assert(chemin(C)),
    fail.
analyse.
```

% **analyse**(R)
R est la liste des actions (sans ordre) d’un parcours complet

```
analyse(R) :-
    debut(X),
    chemin(C,[],[X]),
    filtre_action(C,R).
```

% **chemin**(R,P,L)
R est un parcours complet sachant que P est le parcours deja effectue et L la liste des noeuds susceptibles d’etre immediatement franchis (i.e. qui n’ont pas encore ete franchis mais dont au moins un noeud pere l’a ete).

```
chemin([X|P],P,L) :-
    member(X,L),
    not precede(X,_).
chemin(R,P,L) :-
    member(X,L),
    action(X,_,_),
    supprime(X,L,L1),
    precede(X,Y),
    !,
    ajoute(Y,L1,L2),
    chemin(R,[X|P],L2).
chemin(R,P,L) :-
    member(X,L),
    connecteur(X,et),
    listedes(Y,(precede(Y,X),not member(Y,P)),NIL),
```

```
    NIL = [],
    supprime(X,L,L1),
    precede(X,_),
    !,
    listedes(Z,(precede(X,Z),not member(Z,L)),L2),
    append(L1,L2,L3),
    chemin(R,[X|P],L3).
chemin(R,P,L) :-
    member(X,L),
    connecteur(X,test),
    supprime(X,L,L1),
    precede(X,Y),
    ajoute(Y,L1,L2),
    chemin(R,[X|P],L2).
chemin(R,P,L) :-
    member(X,L),
    connecteur(X,ou),
    supprime(X,L,L1),
    precede(X,Y),
    ajoute(Y,L1,L2),
    chemin(R,[X|P],L2).
```

% **filtre_action**(L,R)

R est la liste des actions de la liste de noeuds L

```
filtre_action([],[]).
filtre_action([X|L],[X|R]) :-
    action(X,_,_),
    !,
    filtre_action(L,R).
filtre_action([X|L],R) :-
    filtre_action(L,R).
```

description de l'ICN

déclaration des actions

% ***action****(identificateur_de_noeud, intervenant, action)*

```
action(a1,bruno,remplir(date)).
action(a2,bruno,remplir(nom)).
action(a3,bruno,remplir(service)).
action(a4,bruno,remplir(telephone)).
action(a5,bruno,remplir(reference)).
action(a6,xavier,remplir(prix)).
action(a7,xavier,remplir(designation)).
action(a8,michel,visa).
action(a9,pierre,visa).
action(a10,francois,visa).
action(a11,xavier,imprimer).
```

déclaration des connecteurs logiques

% ***connecteur****(identificateur_de_noeud, type_de_connexion)*

```
connecteur(et1,et).
connecteur(et2,et).
connecteur(et3,et).
connecteur(et4,et).
connecteur(et5,et).
connecteur(tst1,test).
connecteur(tst2,test).
connecteur(ou,ou).
```

spécification des précédences entre noeuds (définition du treillis orienté)

```
% precede(identificateur_noeud_precedent, identificateur_noeud_suivant)
precede(et1,a1).
precede(et1,a2).
precede(a1,a5).
precede(a5,et2).
precede(et2,a6).
precede(et2,a7).
precede(a6,et4).
precede(a7,et4).
precede(a2,a3).
precede(a3,et3).
precede(et3,et4).
precede(et3,a4).
precede(a4,et5).
precede(et4,a8).
precede(a8,tst1).
precede(tst1,a9).
precede(a9,tst2).
precede(tst2,a10).
precede(tst1,ou).
precede(tst2,ou).
precede(a10,ou).
precede(ou,et5).
precede(et5,a11).
```

expression de sélection de branche pour les connecteurs de type “test”

```
% test(identificateur_noeud_de_test, identificateur_noeud_selectionne)
test(tst1,ou) :-
    valeur(prix,X),
    X < 10000.
test(tst1,a9).
test(tst2,ou) :-
    valeur(prix,X),
    X < 100000.
test(tst2,a10).
```

Activité de l'agent

```
puma :- activite.
% NB : activite executee automatiquement par le reflexe de creation PUMA
activite :-
    init,
    (repeat,
        try((receive(X),traite(X))), % traiter un message
        tache, % tache propre
        (termine ; exception)), % on arrete la boucle lorsque le Workflow est termine
    nl,
    try((termine, write('circulation effectuee normalement'))), % arret normal
    try((exception, write('circulation interrompue par exception'))), % arret par exception
    nl.

% conditions initiales
init :-
    abolish(valeur,2), % elimination des "traces" d'eventuelles executions anterieures
    abolish(histoire,2),
    abolish(termine,0),
    abolish(exception,0),
    abolish(chemin,1),
    analyse, % analyse de l'ICN pour generer les parcours non-bloquants possibles
    myname(X),
    send(X,go). % message de demarrage du Workflow

% tache propre de l'agent
tache :-
    write('.'). % pour bien montrer l'asynchronisme de l'execution
```

traitement des messages

```
% message de demarrage => commencer le Workflow
traite(go) :-
    write('debut de la circulation'),
    nl,
    debut(X), % recherche le premier noeud de l'ICN
    supprconn([],[],X,L), % au cas ou le premier noeud soit un connecteur
    declenche(L). % lance l'execution en parallele de toutes les actions possibles

% action X terminee => declencher les actions rendues possibles
traite(termine(X,R)) :-
    time(T),
    assert(histoire(termine(X),T)), % enregistrement de l'evenement pour l'historique
    nl,
    write('reponse = '),
    write(termine(X,R)),
    action(X,_,remplir(Y)), % enregistrement d'une valeur de champ s'il y a lieu
    assert(valeur(Y,R)),
    fail.
```

```
traite(termine(X,_)) :-
    lances(P),
    termines(A),
    suite(P,A,X,L), % rechercher les nouvelles actions declenchables…
    !,
    declenche(L). % … et les declencher
traite(termine(_,_)) :-
    assert(exception).

% declenchement d’une action ou d’une liste d’actions
declenche(termine) :- % cas particulier de la fin du Workflow
    !,
    assert(termine).
declenche([]).
declenche([X|L]) :-
    !,
    declenche(X),
    declenche(L).
declenche(X) :- % declenchement d’une action = requete asynchrone par message
    action(X,Intervenant,Action),
    myname(Name),
    send(Intervenant,[Name,[X,Action]]),
    write(’requete = ’),
    write([Intervenant,X]),
    time(T),
    assert(histoire(lance(X),T)). % enregistrement du declenchement dans l’historique
```

divers accès à l’historique

```
% liste des actions terminees
termines(L) :-
    listedes([X,Y], histoire(termine(X),Y), L1),
    filtre(L1,L).

% liste des actions lancees (terminees ou non)
lances(L) :-
    listedes([X,Y], histoire(lance(X),Y), L1),
    filtre(L1,L).

% [... [X,_], ...] -> [... X, ...]
filtre([],[]).
filtre([[X,_]|L],[X|R]) :-
    filtre(L,R).

% affichage global de l’historique de la circulation
historique :-
    histoire(X,D),
    write(D),
    write(’ : ’),
```

```
    historique(X),
    nl,
    fail.
historique.

% affichage d'un evenement de l'historique
historique(termine(X)) :-
    !,
    write('action '),
    write(X),
    write(' terminee').
historique(lance(X)) :-
    !,
    write('action '),
    write(X),
    write(' lancee').
historique(X) :-
    write(X).
```

Annexe C : la classe interface

La classe `interface` est utilisée dans le système de réservation de salle (voir Chapitre 7) pour créer un agent représentant une ressource, associé à un objet COOL servant d'interface utilisateur. Cet objet constitue l'interface de contrôle de l'agent, permettant de l'administrer (migration vers une autre machine, destruction) et d'accéder aux services disponibles dans le système multi-agents. Nous présentons cette classe car elle fournit un exemple d'utilisation de la classe `objetSMA`, dont le rôle est d'associer de façon transparente un agent PUMA à tout objet COOL, par simple héritage. Cette approche montre comment on peut définir une sorte de bureau électronique complet dans lequel toutes les applications de bureautique communicante peuvent s'intégrer par le biais du système multi-agents.

L'interface de la classe `objetSMA` : fichier objetSMA.hxx

```
/*****************************************************************\
* objetSMA.hxx
*        projet : puma
*        header de la classe objetSMA
\*****************************************************************/

#include "agent.hxx"

// réflexes déclenchés sur l'agent (si définis dans sa base Prolog) :

#define INITPRED "objetSMA(init)."        // après création
#define REINITPRED "objetSMA(reinit)."    // après migration
#define EXITPRED "objetSMA(exit)."        // avant disparition

// message de requête de migration transmis à l'agent
#define FOLLOWPRED "objetSMA(follow)."
```

```
class objetSMA {
     int agentCreation; // attribut réservé pour gestion interne
protected:
// attributs nécessaires et suffisants pour invoquer l'agent
     agent* agentAddr;       // adresse pour invocation de méthode
     objCap agentMbox;       // adresse pour invocation par message

// constructeur réalisant la création automatique de l'agent
     objetSMA(char* n, char* f);

// méthode d'initialisation de l'agent, après création ou migration
// (termine par un appel au réflexe INITPRED ou REINITPRED)
     void agentInstall(cool* adr);

// méthode de destruction définitive de l'agent
// (commence par un appel au réflexe EXITPRED)
     void agentKill();

// méthode utilitaire : conversion d'une chaîne Prolog en chaîne C
     int list2string(char*, char* = NULL);
};
```

La classe interface

interface.hxx

```
/*****************************************************************\
* interface.hxx
*     projet : réservation intelligente d'une salle de réunion
*     header de la classe interface
\*****************************************************************/

#include "objetSMA.hxx"

class interface : public cool, public objetSMA {
     Rpchar agentName;
// constructeur : 'n' nom de l'agent assosié, 'f' fichier Prolog
     interface(char *n, char *f);
// activité
     virtual void main();
// affichage de la liste des services, et exécution du service requis
     void menu_agent();
// méthode de migration de l'objet et de son agent
     void migration();
};
```

interface.cxx

```
/*****************************************************************\
* interface.cxx
*     projet :reservation intelligente d'une salle de reunion
*     classe 'interface' : interface d'un agent utilisateur
\*****************************************************************/
```

```
#include “interface.hxx”

implement(rPointer,char); // => type Rpchar = pointeur relogeable

extern “C” {
  saisie(char* ptr, int n) {
     char c;
     fflush(stdin);
     while ((c=getchar())!=EOF && c!=’\n’ && c!=’\r’ && n--)
           *ptr++ = c;
     *ptr = ‘\0’;
     return(c != EOF);
  }
  char upcase(char c) {
     return(c >= ‘a’ && c <= ‘z’ ? c - ‘a’ + ‘A’ : c);
  }

interface::interface(char *n, char *f)
// constructeur = appel au constructeur de la classe objetSMA avec
// comme nom symbolique ‘n’ et comme base Prolog initiale le
// fichier ‘f’
: objetSMA(n, f)
{
     agentName = new(this) char[MAXNAMELEN];
     strcpy((char*)agentName, n);
     printf(“INTERFACE DE L’AGENT \”%s\”\n”, n);
}

void interface::main()
// activité = menu de contrôle de l’interface
{
char cmd[2];
agentInstall(this);
do {
     printf(“[%s] I.nvoquer l’agent - D.emenager - T.erminer : “,
         (char*) agentName);
     saisie(cmd, 1);
     switch(*cmd = upcase(*cmd)) {
     case ‘I’ :
         menu_agent();
         break;
     case ‘D’ :
         migration();
         break;
     case ‘T’ :
         break;
} while (*cmd != ‘T’);
agentKill();
printf(“disparition de l’agent \”%s\”.\n”, (char*) agentName);
delete((char*) agentName);
}

void interface::migration()
// menu de migration vers un autre objet COOL, reperé par son nom.
```

```
// (l’objet de destination doit faire appel à objectReceive())
{
objCap destCap;
msgDesc msg;
char nom[MAXNAMELEN];

printf(“migrer vers : “);
saisie(nom, MAXNAMELEN-1);
if (nameLookup(&destCap, nom) == C_OK) {
     msg.flags = 0;
     msg.msg = “objetSMA(object)”;
     msg.msgSize = strlen(msg.msg)+1;
     msg.obj = (cool*) this;
     objectSend(&destCap, &msg);
     printf(“erreur de migration\n”);
}
else
     printf(“demenagement impossible : destination injoignable.\n”);
}

void interface::menu_agent()
// invocation de l’agent pour connaître les services proposés
// et affichage du menu resultant
{
char x[MAXPREDLEN], y[MAXPREDLEN];
int etatPuma;

agentAddr->puma_P(); // réquisition du noyau Prolog de l’agent
// invocation du but Prolog interface(X,Y)
if ((etatPuma = agentAddr->puma_call(“interface(X,Y).”)) == PFAIL) {
     agentAddr->puma_V();
     printf(“l’agent %s ne propose pas de service a l’utilisateur.\n”,
         (char*)agentName);
}
else { // affichage de la liste des (X,Y) tels que interface(X,Y)
     do {
         agentAddr->puma_unif(“X”, x);
         agentAddr->puma_unif(“Y”, y);
         list2string(y);
         printf(“%s : %s\n”, x, y);
         etatPuma = agentAddr->puma_recall(); // backtrack
     } while (etatPuma != PFAIL);
     agentAddr->puma_V(); // libération du noyau Prolog de l’agent
     printf(“requete : “);
     saisie(x, MAXPREDLEN-1);
     if (*x) { // invocation du service
         agentAddr->puma_P();
         sprintf(y, “interface(%s).”, x);
         if (agentAddr->puma_call(y) == PERROR)
            printf(“erreur d’invocation !\n”);
         agentAddr->puma_V();
     }
}
}
```

Annexe D : la résolution de contraintes

Le langage

Le langage d'expression de contraintes est à la base de notre mécanisme de négociation de service entre agents. Il permet d'exprimer des contraintes sur les caractéristiques d'un service recherché (point de vue du client) ou proposé (côté serveur). Pour des commodités d'écriture, un certain nombre d'opérateurs Prolog ont été ajoutés :

```
:- op(150, yfx, [ before, after ]). % operateurs generiques de precedence
:- op(150,  fx, [ @, \@, :=, >=, =< ]). % contraintes
```

- opérateurs de précédence générique before et after. Quelques règles les définissent par défaut, mais on peut étendre cette définition pour des types de caractéristique particuliers. Les deux procédures qui en résultent sont exécutées pour déterminer respectivement les contraintes "inférieur" et "supérieur".

```
% operateur de comparaison 'before'
% pour les dates Jour/Mois/Annee
J1/M/A before J2/M/A :-
      J1 before J2,
      !.
_/M1/A before _/M2/A :-
      M1 before M2,
      !.
_/_/A1 before _/_/A2 :-
      A1 before A2,
      !.
% cas general
X before Y :-
      X @=< Y.
```

• symboles de contraintes

[identificateur_caracteristique, @ [v_1, v_2…v_i], \@ [v'_1, v'_2…v'_j], >= b_{inf}, =< b_{sup}, maxi]	
@	la valeur de la caractéristique doit être égale à v_1, v_2 …ou v_i
\@	la valeur de la caractéristique doit être différente de v'_1, v'_2 …et v'_j
>=	la valeur de la caractéristique doit être supérieure ou égale à b_{inf}
=<	la valeur de la caractéristique doit être inférieure ou égale à b_{sup}
maxi	la valeur doit être la plus haute possible en respectant l'ensemble des contraintes
mini	la valeur doit être la plus faible possible en respectant l'ensemble des contraintes

Chaque requête de service comprend un identificateur de service et une liste qui contient :

- une liste de contraintes pour chaque caractéristique (s'il y a lieu), au format [identificateur_caractéristique, contrainte_1, contrainte_2, … contrainte_n] (voir exemple ci-dessus) ;
- une liste particulière permettant de fournir la fonction de calcul de la satisfaction (le cas échéant), au format [satisfaction(Resultat), expression_de_calcul].

exemple :
[… [satisfaction(X), X is 1 / (1 + (&chaises-10)/&chaises + (&tables-6)/&tables)], …]
L'opérateur unaire & permet d'indiquer que l'on veut substituer l'identificateur de caractéristique qui suit par sa valeur résolue (chaises et tables désignent des caractéristiques).

La résolution

Le processus de résolution globale comprend plusieurs étapes successives :

(1) Toutes les contraintes sont d'abord recomposées par la procédure fusion_contraintes/3. Cela permet de simplifier l'expression globale et de détecter des incompatibilités génériques. Les deux opérateurs de précédence générique (before, after) interviennent lors de cette étape.

(2) La résolution effectuée par resoudre_contraintes/3 consiste à engendrer une valuation complète des caractéristiques à partir de la liste des contraintes simplifiée au cours de l'étape précédente, et d'une liste d'éventuelles de valuations par défaut. Le format de la valuation est de la forme [… [caractéristique, valeur], …]. La résolution des valeurs des caractéristiques est définie de façon générique par un ensemble de règles, mais elle peut être complétée pour des caractéristiques particulières en ajoutant des clauses à la procédure resoudre_caracteristique/3.

(3) L'évaluation de la satisfaction est réalisée par evaluer_satisfaction/3 à partir des caractéristiques valuées et de l'expression de calcul fournies.

Fichier Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% util_contraintes
%       bibliotheque de procedures de resolution de contraintes
%       evaluation de la satisfaction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% fusion_contraintes/3
% (composer_contrainte/2 , ajouter_contraintes/3, ajouter_une_contrainte/3)
% resoudre_contraintes/3, resoudre_caracteristique/3, (resoudre_une_caract1/2)
% evaluer_satisfaction/3, (substituer_variables/3)

%==================================================================
% composition / simplification de contraintes d'origine avec des contraintes supplementaires
%==================================================================

% composition de 2 contraintes. Verifie la compatibilite et simplifie
% Echoue en cas d'incompatibilite
%       composer_contrainte(C1, CompC2, NouvC1, NouvC2)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% mini, maxi / parmi, sauf, >=, =<
composer_contrainte(C, mini, C, mini).
composer_contrainte(C, maxi, C, maxi).

% parmi/parmi
composer_contrainte(@ L1, @ L2, true, @ L3) :-
    bagof(X, (member(X, L1), member(X, L2)), L3).

% parmi/sauf
composer_contrainte(@ L1, \@ L2, true, @ L3) :-
    bagof(X, (member(X, L1), not member(X, L2)), L3).
composer_contrainte(\@ L1, @ L2, true, @ L3) :-
    bagof(X, (member(X, L2), not member(X, L1)), L3).

% sauf/sauf
composer_contrainte(\@ L1, \@ L2, true, @ L3) :-
    listedes(X, (member(X, L1), not member(X, L2)), L),
    append(L, L2, L3).

% inferieur/inferieur
composer_contrainte(=< X, =< Y, =< X, true) :-
    X before Y.
composer_contrainte(=< X, =< Y, true, =< Y) :-
    X after Y.

% inferieur/superieur
composer_contrainte(=< X, >= X, true, @ [X]).
```

```
composer_contrainte(=< X, >= Y, =< X, >= Y) :-
    not (X before Y).
composer_contrainte(>= X, =< X, true, @ [X]).
composer_contrainte(>= X, =< Y, >= X, =< Y) :-
    not (X after Y).

% superieur/superieur
composer_contrainte(>= X, >= Y, >= X, true) :-
    X after Y,
    !.
composer_contrainte(>= X, >= Y, true, >= Y) :-
    X before Y.

% parmi/inferieur
composer_contrainte(@ L1, =< X, true, @ L2) :-
    bagof(Y, (member(Y, L1), Y before X), L2).
composer_contrainte(=< X, @ L1, true, @ L2) :-
    bagof(Y, (member(Y, L1), Y before X), L2).

% parmi/superieur
composer_contrainte(@ L1, >= X, true, @ L2) :-
    bagof(Y, (member(Y, L1), Y after X), L2).
composer_contrainte(>= X, @ L1, true, @ L2) :-
    bagof(Y, (member(Y, L1), Y after X), L2).

% sauf/inferieur
composer_contrainte(\@ L1, =< X, \@ L2, =< X) :-
    bagof(Y, (member(Y, L1), Y before X), L2).
composer_contrainte(\@ L, =< X, true, =< X) :-
    not bagof(Y, (member(Y, L), Y before X), _).
composer_contrainte(=< X, \@ L1, =< X, \@ L2) :-
    bagof(Y, (member(Y, L1), Y before X), L2).
composer_contrainte(=< X, \@ L, =< X, true) :-
    not bagof(Y, (member(Y, L), Y before X), _).

% sauf/superieur
composer_contrainte(\@ L1, >= X, \@ L2, >= X) :-
    bagof(Y, (member(Y, L1), Y after X), L2).
composer_contrainte(\@ L, >= X, true, >= X) :-
    not bagof(Y, (member(Y, L), Y after X), _).
composer_contrainte(>= X, \@ L1, >= X, \@ L2) :-
    bagof(Y, (member(Y, L1), Y after X), L2).
composer_contrainte(>= X, \@ L, >= X, true) :-
    not bagof(Y, (member(Y, L), Y after X), _).


% fusion d'une liste de [caracteristique|contraintes] avec une autre
% echoue en cas d'incompatibilite des nouvelles contraintes avec les anciennes
%       fusion_contraintes(+Nouv_car-contr, +Anc_car-contr, ?Resultat)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fusion_contraintes([], L, L).
fusion_contraintes([ [ Carac , Prg ] | Cdr], Old, New) :-
    not atom(Carac),
    !,
    fusion_contraintes(Cdr, [[ Carac, Prg ] | Old ], New).
fusion_contraintes([ [ Carac | Contr ] | Cdr], Old, New) :-
    retire([Carac | OldContr], Old, Old2),
    !,
    ajouter_contraintes(Contr, OldContr, L),
    fusion_contraintes(Cdr, [[ Carac | L] | Old2 ], New).
fusion_contraintes([ [ Carac | Contr ] | Cdr], Old, New) :-
    !,
    ajouter_contraintes(Contr, [], L),
    fusion_contraintes(Cdr, [[Carac | L] | Old], New).


% fusion d'une liste de contraintes avec une autre
% echoue en cas d'incompatibilite
%       ajouter_contraintes(Nouvelles_contraintes, Anciennes_contraintes, Resultat)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ajouter_contraintes([], L, L).
ajouter_contraintes([Car | Cdr], L1, L2) :-
    ajouter_une_contrainte(Car, L1, L),
    ajouter_contraintes(Cdr, L, L2).


% composition d'une contrainte avec une liste de contraintes
% echoue en cas d'incompatibilite
%       ajouter_une_contrainte(Nouvelle_contrainte, Contraintes_a_composer, Resultat)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ajouter_une_contrainte(C, [], [C]) :-
    !.
ajouter_une_contrainte(C, [Car | Cdr], R) :-
    composer_contrainte(Car, C, true, C2),
    !,
    ajouter_une_contrainte(C2, Cdr, R).
ajouter_une_contrainte(C, [Car | Cdr], [Car2 | Cdr]) :-
    composer_contrainte(Car, C, Car2, true),
    !.
ajouter_une_contrainte(C, [Car | Cdr], [Car2 | Cdr2]) :-
    composer_contrainte(Car, C, Car2, C2),
    !,
    ajouter_une_contrainte(C2, Cdr, Cdr2).
```

```
%===============================================================
% resolution des contraintes : recherche d'une valeur pour chaque caracteristique
%===============================================================

% resolution (valuation) des contraintes en utilisant des contraintes par defaut
% si necessaire. Echoue si la valuation est impossible ou incomplete.
% Les expressions de type satisfaction(X) sont ignorees
%       resoudre_contraintes(+carac/contr, +contrParDefaut, -Valuations)
%                   [[Car|Contr]|...], [[Car|Contr]|...], [[Car,Val],...[Satisfaction(X),Expr] |...]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

resoudre_contraintes([], _, []).
resoudre_contraintes([[satisfaction(_), _] | Cdr], Defauts, R) :-
    !,
    resoudre_contraintes(Cdr, Defauts, R).
resoudre_contraintes([[Carac | Contr] | Cdr], Defauts, [[Carac, Val] | R]) :-
    resoudre_caracteristique(Carac, Contr, Val),
    !,
    resoudre_contraintes(Cdr, Defauts, R).
resoudre_contraintes([[Carac | Contr] | Cdr], Defauts, [[Carac, Val] | R]) :-
    member([Carac | ContrDefaut], Defauts),
    ajouter_contraintes(Contr, ContrDefaut, Contr2),
    resoudre_caracteristique(Carac, Contr2, Val),
    !,
    resoudre_contraintes(Cdr, Defauts, R).

% resolution des contraintes pour une caracteristique (valuation)
% regles par defaut, generiques (independantes de la caracteristique);
% a enrichir pour les cas generiquement indecidables.
%       resoudre_caracteristique(+Carac, +Contr, -Val)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

resoudre_caracteristique(_, Contr, R) :-
    sort(Contr, OrdContr), % tri prealable pour diminuer le nbre de combinaisons
    resoudre_une_caract1(OrdContr, R).

% =< / maxi
resoudre_une_caract1([maxi, =< R], R).
resoudre_une_caract1([maxi, =< R, >= _], R).
resoudre_une_caract1([maxi, =< R, \@ E], R) :-
    not member(R, E).
resoudre_une_caract1([maxi, =< R, >= _, \@ E], R) :-
    not member(R, E).

% >= / mini
resoudre_une_caract1([mini, >= R], R).
resoudre_une_caract1([mini, =< _, >= R], R).
```

```
resoudre_une_caract1([mini, >= R, \@ E], R) :-
    not member(R, E).
resoudre_une_caract1([mini, =< _, >= R, \@ E], R) :-
    not member(R, E).

% @
resoudre_une_caract1([@ [X]], X).
resoudre_une_caract1([mini, @ E], R) :-
    minimum(E, R).
resoudre_une_caract1([maxi, @ E], R) :-
    maximum(E, R).

%=============================================================
% evaluation de la satisfaction pour une offre
%=================================

% evaluation de la satisfaction. Echoue si la satisfaction ne peut etre evaluee
%       evaluer_satisfaction(Offre, Defauts, Satisfaction)
%                               [[Car,Val],...[satisfaction(X),prg]], <idem>, Satisfaction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

evaluer_satisfaction(Vars, _, Satisfaction) :-
    member([satisfaction(S), Prg], Vars),
    substituer_variables(Vars, Prg, Prg2, PlVars),
    bagof(S, PlVars^Prg2, [Satisfaction]),
    !.
evaluer_satisfaction(Vars, Defauts, Satisfaction) :-
    not member([satisfaction(_), _], Vars),
    member([satisfaction(S), Prg], Defauts),
    substituer_variables(Vars, Prg, Prg2, PlVars),
    bagof(S, PlVars^Prg2, [Satisfaction]),
    !.

% Substitue les variables (&...) contenues dans le terme Prg par leur valeur
% et etablit la liste des variables Prolog presentes dans Prg.
% Echoue si une variable (&...) n'a pas de valeur dans Vars.
%       substituer_variables(Vars, Prg, Res, PrologVars)
%                               [[Var,Val]|...], [T1|...], [T1'|...], [X | ...]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

substituer_variables(_, X, X, []) :-
    atomic(X),
    !.
substituer_variables(_, X, X, [X]) :-
    var(X),
    !.
substituer_variables(_,[],[],[]) :-
    !.
```

```
substituer_variables(Vars, [Car|Cdr], [R1 | R2], L) :-
    !,
    substituer_variables(Vars, Car, R1, L1),
    substituer_variables(Vars, Cdr, R2, L2),
    merge(L1, L2, L).
substituer_variables(Vars, &Var, Val, []) :-
    member([Var, Val], Vars),
    !.
substituer_variables(_, &Var, _, _) :-
    !,
    fail.
substituer_variables(Vars, Expr, Res, PlVars) :-
    Expr =.. L1,
    substituer_variables(Vars, L1, L2, PlVars),
    Res =.. L2.
```