
PROJECTE FINAL DE CARRERA



UNIVERSITAT
ROVIRA I VIRGILI



Avaluació d'entorns de desenvolupament de Sistemes Multiagent

David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

1. DESCRIPCIÓ I OBJECTIUS	3
2. AGENTS	4
2.1. DEFINICIÓ.....	4
2.2. ATRIBUTS D'AGENTS.....	6
2.3. ARQUITECTURES D'AGENTS.....	8
2.3.1. <i>Deliberatives</i>	8
2.3.1.1. Arquitectures BDI.....	9
2.3.2. <i>Reactives</i>	11
2.3.3. <i>Híbrides</i>	12
2.3.3.1. <i>InteRRaP</i>	13
2.3.3.2. <i>TouringMachines</i>	14
2.4. TIPUS D'AGENTS.....	15
2.4.1. <i>Agents d'informació/Internet</i>	15
2.4.2. <i>Agents d'interfície</i>	16
2.4.3. <i>Agents de col·laboració</i>	17
2.4.4. <i>Mòbils</i>	19
2.4.5. <i>Agents reactius</i>	20
2.4.6. <i>Híbrids</i>	21
2.4.7. <i>Heterogenis</i>	22
3. SISTEMES MULTIAGENT	24
3.1. COMUNICACIÓ.....	24
3.2. LENGUATGES DE COMUNICACIÓ (ACL) ESTÀNDARDS.....	25
3.2.1. <i>Llenguatge de comunicació de la FIPA</i>	25
3.2.2. <i>Knowledge Query and Manipulation Language (KQML)</i>	27
3.2.2.1. <i>Knowledge Interchange Format (KIF)</i>	31
3.3. LENGUATGES DE COMUNICACIÓ (ACL) AD HOC.....	32
3.4. NEGOCIACIÓ I COOPERACIÓ EN SMA	33
3.4.1. <i>Teoria de Jocs</i>	33
3.4.2. <i>Contract Net Protocol (CNP)</i>	34
3.4.3. <i>Distributed Problem Solve (DPS)</i>	34
4. APLICACIONS DELS SISTEMES MULTIAGENT	35
5. LLISTA D'EINES DE DESENVOLUPAMENT SMA	37
6. REFERÈNCIES	41
7. ESTUDI ESPECÍFIC D'ALGUNES EINES	42
7.1. ENTORNS ESCOLLITS.....	42
7.2. ENTORNS NO ESCOLLITS.....	42
7.2.1. <i>AgentBuilder</i>	43
7.2.2. <i>DECAF</i>	50
7.2.3. <i>FIPA SMART</i>	52
8. ESTUDI DE JACK INTELLIGENT AGENTS	53
9. ESTUDI DE JATLITE BETA	83
10. ESTUDI DE JAFMAS	116
11. ESTUDI DE ZEUS	155
12. ESTUDI DE KAFKA	212
13. ESTUDI COMPARATIU.....	241

1. DESCRIPCIÓ I OBJECTIUS

El món de la Intel·ligència Artificial està constantment evolucionant. Per altra banda, la tecnologia cada cop permet de fer coses que abans no eren possibles o eren impracticables.

Una de les revolucions més importants dels últims anys és l'ús de la transmissió de dades a través de xarxes electròniques. Aquesta transmissió cada cop més ràpida i segura, fa pensar als investigadors en distribuir processos executant-se concurrentment, i fer que la comunicació i la cooperació entre ells, siguin la base de noves teories. Una d'aquestes teories és la dels *agents*.

Aquesta àrea de treball/recerca encara no té la maduresa d'altres tècniques clàssiques de IA. Per això, encara que no hi ha una definició àmpliament extesa, hi ha una entesa general d'allò que no són els agents. Aquest document intenta descriure que són els agents i, a la vegada, es donen casos que podrien entendre's com agents i en canvi no ho són.

L'ús d'agents cada cop és més important. Tot i que les funcions que poden realitzar actualment són limitades, les possibilitats que hom pot veure-hi en un futur són molt grans. Podríem pensar en agents que fessin tot allò que els hi demanéssim de forma autònoma, eficient, cooperant/aprenent amb l'usuari, i en qualsevol àmbit de treball. Encara que som lluny d'aquestes expectatives, la proliferació diària de noves eines, de nous productes cada cop més perfeccionats, fa pensar en un futur brillant per aquesta nova eina de treball.

En aquest treball, es pot trobar una síntesi de les investigacions en aquest camp, i una relació de les eines que es poden trobar a Internet actualment. S'ha aprofundit en algunes d'aquestes eines i s'han estudiat per donar una especificació de les funcions que realitzen i com les realitzen.

Aquest PFC inclou dos aspectes:

A) Fer una cerca exhaustiva (via Internet) d'entorns que faciliten la construcció de sistemes multiagent¹.

B) Fer una avaluació comparativa d'un subconjunt seleccionat d'aquests entorns. S'han avaluat aspectes tals com els recursos necessaris en cada entorn, la facilitat d'aprenentatge de les eines que proporciona cada sistema, la facilitat de desenvolupament de cada agent i del sistema sencer, les eines de comunicació dels agents que es disposa, etc.

¹ No hi ha una normativa clara respecte l'ús de multi-agent o multiagent. Segons l'acord de la Secció Filològica del 15 de març de 1996 de l'Institut d'Estudis Catalans, referent a "*l'ús de guionet en l'escriptura de mots formats per composició o per prefixació*", del segon paràgraf es desprèn que el mot correcte en aquest cas és **multiagent**.

2. AGENTS

És difícil definir què és un agent. Aquest concepte genera confusió entre la comunitat científica i ha esdevingut un tema de debat durant els últims anys.

Encara que el terme agent ha estat utilitzat sense moderació per a referir-se a qualsevol sistema de software que té atributs d'intel·ligència, autonomia, percepció o interacció amb l'usuari, no hi ha consens sobre una definició estàndard. Els científics han proposat varies definicions pel terme i encara ho estan fent. Les diferents definicions que es poden trobar es descriuen en la subsecció següent.

En realitat la paraula “agent” és un terme que engloba una àrea de recerca i desenvolupament heterogeni.

2.1. Definició

Veurem diferents definicions proposades per alguns investigadors.

Donant una visió idealitzada, Nicholas Negroponte i Alan Kay [1], diuen:

Un agent pot ser considerat un programa assistent personal (*personal software assistant*) amb autoritat delegada per l'usuari, per fer tasques que a l'usuari li són difícils o impossibles de realitzar. Aquests programes simulen una relació humana per fer alguna cosa que cap altra persona podria fer per tu.

Per Hyacinth S. Nwana de BT i, Michael Wooldridge de Queen Mary College [7], un agent és un component de software i/o hardware que és capaç d'actuar per complir tasques en representació d'un usuari. Quan ha de realitzar alguna tasca, cal donar-li aquesta informació en expressions, meta-expressions, o classes. Aquesta informació cobreix diferents tipus d'agents i cal triar de quin tipus es tracta.

Segons Pattie Maes de l'MIT [2],

Un agent és un sistema computacional que habita en un entorn complex i dinàmic. L'agent pot sentir i actuar autònomament en el seu entorn, i té un conjunt d'objectius o motivacions que ell intenta d'aconseguir a través de les seves accions.

Per la *Foundation for Intelligent Physical Agents* (FIPA) [1],

Un agent és una entitat que resideix en entorns, on interpreta dades de sensors reflectint events d'aquest entorn, i executa comandes que produeixen certs efectes en l'entorn. Un agent pot ser purament software o hardware. En el darrer cas, es necessita gran quantitat de software per a fer del hardware un agent.

D'aquestes definicions hom pot veure que no hi ha una definició estàndard del terme agent, se sap què no ho és. Aquestes definicions poden portar a confusions o errors, i considerar agents a altres sistemes de software o hardware. Així, si llegim la definició de la FIPA, podríem considerar com agent un subrutina que efectui qualsevol acció responnent a uns estímuls, però això no és cap agent. També pot haver exemples concrets d'aplicació que no estan explícitament considerats en les definicions, però que tothom considera com agents (per exemple el cas de robots mòbils, amb una autonomia determinada).

Davant d'aquesta falta de definició, molts investigadors inventen sinònims per tal d'explicar-ne el significat, i això encara fa créixer la confusió. Alguns d'aquests sinònims són: knowbots (robots basats en el coneixement), softbots (software robot), taskbots (robots basats en tasques), robots, agents personals, agents autònoms, i assistents personals. Els robots són dispositius que habiten en un món real; si enlloc d'això, habiten en xarxes de computadors, s'anomenen softbots; si tenen assignades tasques específiques, s'anomenen taskbots; i robots autònoms són agents mòbils o robots els quals operen en un entorn dinàmic i incert. Els knowbots i els assistents personals, són agents que són modelats per regles i tenen coneixement de dominis concrets.

2.2. Atributs d'agents

Com sembla que no es pot trobar una definició globalment acceptada d'allò que és un agent, la via més usada per definir agents consisteix a donar una llista de propietats o atributs necessaris per tal que un sistema hardware o més sovint software, sigui considerat un agent:

A) *Autonomia.*

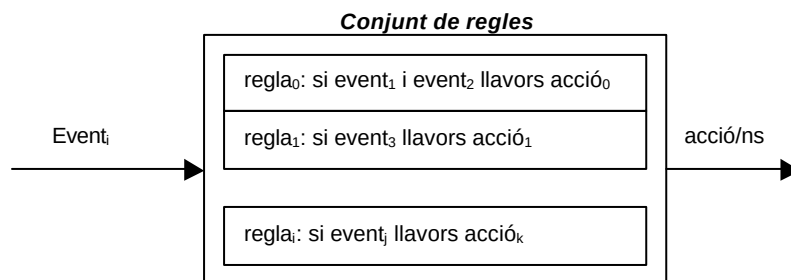
Els agents han d'operar sense la intervenció directa dels humans o altres, i tenir algun tipus de control sobre les pròpies accions i l'estat intern [3]. Això requereix aspectes com accions periòdiques, execucions espontànies i iniciativa, i que cada agent ha de poder prendre accions preventives o independents que beneficiaran l'usuari [1].

B) *Sociabilitat/cooperació.*

Els agents interactuen amb altres agents i/o humans a través d'algun tipus de llenguatge de comunicació [3]. La interacció entre l'usuari i l'agent pot ser descrita com un contracte de col·laboració on l'usuari d'unes especificacions que hauran de ser portades a terme per l'agent. Per exemple, una comanda d'un determinat producte a un agent, el qual controla la fabricació d'aquest producte. La interacció entre agents és un mecanisme pel qual els agents intercanvien el seu coneixement, les seves creences i els seus plans per treballar junts i solucionar grans problemes que es troben més enllà de les seves possibilitats.

C) *Reactivitat.*

Els agents perceben el seu entorn i respon en un temps raonable als canvis detectats [3]. Les accions es realitzen quan es compleixen les precondicions de les regles, que donen lloc a l'execució d'aquestes, o per l'execució de plans ja predissenyats.



D) *Pro-activitat o capacitat de prendre iniciativa.*

Els agents no només han de respondre al seu entorn, sinó que han d'exhibir algun tipus de comportament encaminat a assolir algun tipus d'objectiu prenent la iniciativa [3].

Altres atributs:

E) *Mobilitat.*

Els agents poden moure's a altres entorns a través d'una xarxa electrònica [3]. Poden traslladar dades amb instruccions que poden ser executades remotament [1].

F) *Continuïtat temporal*

Els agents estan contínuament executant processos, no només un procés que s'acaba [1].

G) *Veracitat*

És l'assumpció que un agent no comunicarà informació falsa premeditadament [3].

H) *Benevolència.*

És l'assumpció que els agents no tenen objectius conflictius, i que cada agent sempre intentarà fer allò pel qual és requerit [3].

I) *Racionalitat.*

És l'assumpció que un agent actuarà per tal d'aconseguir el seu objectiu. No realitzarà una acció si les condicions per realitzar-la no es compleixen [3].

2.3. Arquitectures d'agents

Fins ara s'ha vist una visió d'allò que entenem per agent i les propietats que poden tenir. En aquesta secció es veuen tècniques per construir sistemes de computadors que satisfan les propietats dels agents, especificades teòricament.

Segons Pattie Maes d'MIT [3], una arquitectura d'agents es defineix com:

“Metodologia particular per a construir agents. Especifica com l'agent pot ser descompost en la construcció d'un conjunt de mòduls i com aquests haurien d'interactuar. El conjunt total de mòduls i les seves interaccions ha de proporcionar una resposta a la pregunta de: Com les dades dels sensors i l'estat actual de l'agent determinen les accions i el nou estat resultant. Una arquitectura es compon de tècniques i algorismes que suporten aquesta metodologia”

Hi ha diferents arquitectures, però es poden classificar en tres tipus: *deliberatives*, *reactives* i *híbrides*. En les subseccions següents es veuran en detall.

2.3.1. Deliberatives

Les arquitectures deliberatives es basen en la hipòtesi dels símbols físics [3] (Newell/Simon, 1976). Un sistema físic de símbols és definit per a ser un conjunt d'entitats físiques (símbols) que poden ser combinades per a formar estructures.

Aquestes estructures poden executar processos que operen sobre els símbols, segons un conjunt d'instruccions. Aquesta estructura és capaç de generar accions intel·ligents.

La base d'aquesta teoria és la manipulació de raonament lògic: hem de fer que un agent faci alguna teoria (propietat) que sigui prou simple com per donar la seva representació lògica.

Aquesta teoria presenta dos problemes: *i)* el problema de traduir el món real a una bona descripció simbòlica, i *ii)* el problema de com representem la informació d'un complex món real en entitats i processos, i com els agents hi treballen en un temps raonable.

Els científics han desenvolupat diverses tècniques d'IA per desenvolupar eines deliberatives però el seu domini és l'aplicació de les arquitectures més clàssiques en IA, sobre les que s'han desenvolupat molts mètodes i tècniques.

2.3.1.1. Arquitectures BDI

L'estat d'un agent està representat per cinc estructures, tres de bàsiques: *creences* (*beliefs*), *desitjos* (*desires*) i *intencions* (*intentions*), i dues més que calen per dur a terme les tres primeres: *objectius* i *plans*.

- Les creences són el model del domini. És tot allò que coneix (que és cert del món). Aquestes creences també poden ser creences sobre altres agents.
- Els desitjos, a partir de les creences, donen algun tipus d'ordre de preferència entre estats o situacions que l'agent voldria que s'esdevinguessin.
- Els objectius són un subconjunt dels desitjos sobre els que l'agent pot intentar fer accions per obtenir-los. Han de ser coherents, realitzables i no contradictoris.
- Les intencions són un subconjunt dels objectius que l'agent decideix d'intentar de forma activa.
- Els plans són la seqüència d'accions, que d'acord amb les creences, serviran per assolir una determinada intenció.

La implementació d'aquesta arquitectura pot fer-se de moltes maneres, una forma possible és de la següent (vista a IA2): *arquitectura BDI Global*. (v. Figura 2.2)

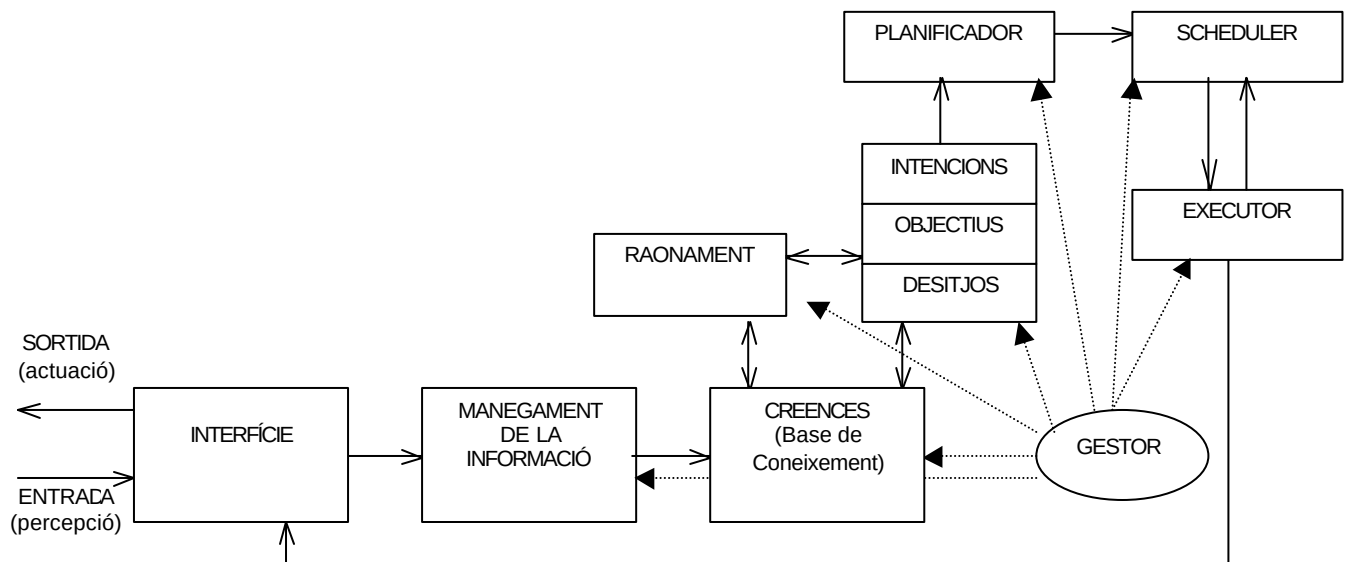


Figura 2.2: Arquitectura BDI Global (Extret d'apunts d'IA2)

La interfície és el pont entre el món real i el sistema. Per aquesta es fa l'entrada d'informació al sistema, i s'interpreten les accions a realitzar, i es dona la sortida corresponent.

El manegament de la informació és necessari per traduir/interpretar la informació que li arriba de la interfície. El manegament interacciona amb les creences, així cal un llenguatge per tal de guardar la informació de manera adequada.

Les creences (estructures de dades), formen la base de coneixement del sistema.

El mòdul de raonament genera, a partir de les creences, els desitjos, els objectius i les intencions.

El planificador construeix els plans, a partir de les intencions. El pla és una seqüència d'accions a prendre per assolir un fi determinat. Els plans s'han de generar dinàmicament a mida que les intencions vagin canviant.

L'scheduler ha de determinar quan/com s'executarà cada acció. La forma d'execució té (quan/com) en compte mesures com temps òptim, temps límit, recursos necessaris, etc. de manera que es compleixin unes especificacions. L'scheduler pot canviar l'ordre d'execució d'un pla, si aquest pot veure's optimitzat.

L'executor du a terme les accions, monitoritzant-les. Si les accions no compleixen les condicions determinades per l'scheduler (passar el límit de temps, passar-se de recursos, etc.) es retorna el pla a l'scheduler.

2.3.2. Reactives

Com s'ha vist, l'arquitectura deliberativa té algunes mancances. Per solucionar aquests problemes, es van desenvolupar l'arquitectura anomenada reactiva. L'arquitectura reactiva no treballa amb cap tipus de model simbòlic i no usa raonament simbòlic.

ARQUITECTURA DE SUBSUMPCIÓ (BROOKS, 1986)

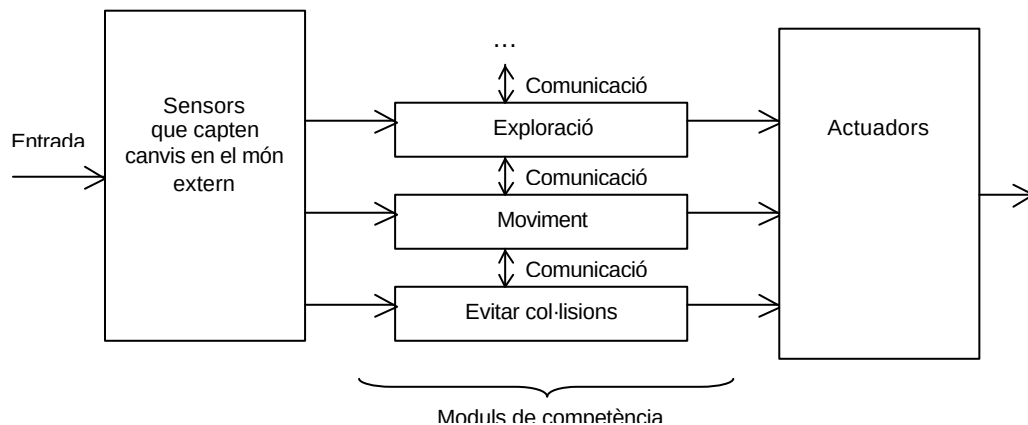


Figura 2.3: Arquitectura de subsumpció (extret de [5])

És una arquitectura horitzontal, doncs totes les dades arriben als mòduls en paral·lel (a la vegada).

Els mòduls de competència treballen de manera independent i asíncrona.

Cada mòdul està implementat amb un autòmat d'estats finits que s'activa quan el senyal d'entrada supera un cert nivell.

Entre els mòduls hi ha una comunicació per intercanviar informació quan és necessari.

La posició dels mòduls està pensada de manera jeràrquica ordenada per la complexitat de les operacions que han de realitzar, de manera que un nivell i èssim, sap tot allò que ha succeït en els nivells inferiors, $1 \leq i \leq n$.

2.3.3. Híbrides

Alguns científics han suggerit que molts entorns d'aplicació no són del tot reactius, o deliberatius. Així ha sorgit la idea de sistemes híbrids que intenten casar les dues aproximacions clàssiques.

Una aproximació és construir un agent amb dos (o més) subsistemes: un *deliberatiu* i un *reactiu*. El deliberatiu té coneixement del model del món i pot fer-ne planificació i prendre decisions. El reactiu és capaç de manegar events que es donen en l'entorn sense utilitzar un raonament complex. Normalment, el reactiu té preferència sobre el deliberatiu, i això pot proporcionar una resposta ràpida.

No hi ha massa implementacions d'aquests sistemes. Els exemples més importants són **InteRRaP** i **TouringMachines**.

2.3.3.1. InteRRaP

Aquest model d'arquitectura va ser dissenyat per Müller et al. del *German Artificial Intelligence Research Institute* (<http://www.dfki.uni-sb.de/~pischel/interrap.html>) el 1994.

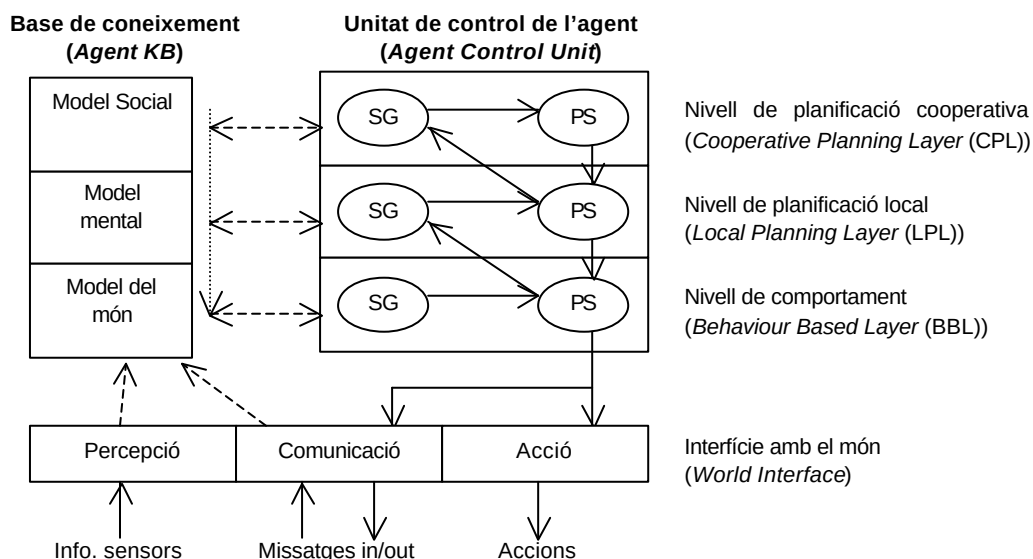


Figura 2.4: Arquitectura híbrida definida per InteRRaP

L'arquitectura modular d'InteRRaP (v. Figura 2.4) està subdividida en dos mòduls verticals: la *base de coneixement* i la *unitat de control*. El primer conté els nivells de la base de coneixement; el segon, conté alguns components de control, que interactuen amb llurs nivells de coneixement. En el nivell més baix tenim la *interfície amb el món*, que com diu el seu nom, és el pont entre l'agent i el seu entorn; té les funcions de realitzar accions (actuadors), de comunicar-se (enviar/rebre informació), i percebre (sensors).

Tant la base de coneixement com la unitat de control de l'agent, estan dividides en nivells.

La unitat de control de l'agent està formada per tres nivells: nivell de comportament (*behaviour-based layer* (BBL)), nivell de planificació local (*local planning layer* (LPL)), i el nivell de planificació cooperativa (*cooperative planning layer* (CPL)).

El BBL implementa la part reactiva del sistema. Aquest mòdul manipula un conjunt de patrons de comportament (*patterns of behaviour* (PoB)). Cada PoB és una estructura en forma de regla: *precondició* → *acció* → *postcondició*. Aquestes accions (*body*) d'un PoB poden fer crides directament a la interfície amb el món, o cridar a un nivell superior per tal de fer una planificació.

El nivell immediatament superior al BBL és el CPL, o nivell de planificació local. Aquest nivell és capaç de generar plans² donant resposta als requeriments que s'envien del nivell inferior (BBL). La base de coneixement d'aquest nivell (nivell mental) conté un conjunt de plans realitzats (llibreria de plans).

El nivell superior és el de cooperació (CPL). Aquest nivell és capaç de generar plans compartits que satisfacin objectius d'un conjunt d'agents responen a la crida que es fa

² Ens referim a un pla com una seqüència d'accions que fan possible un fi determinat.

del nivell inferior (LPL). La base de coneixement d'aquest nivell, també conté una llibreria de plans.

Cada nivell de la unitat de control està format per dos processos: SG i PS. El procés SG realitza les funcions de reconeixement de situacions i activació d'objectius; el procés PS és el procés de planificació, scheduling i execució.

El sistema rep una entrada a través de la interfície del món, i normalment hi ha un canvi en el model del món. Com a resultat d'aquest/s canvi/s, varis PoB poden ser activats. Com a resultat d'aquesta execució, el LPL i CPL poden respondre generant plans locals de l'agent, o generant plans compartits entre agents respectivament per tal d'aconseguir els objectius de l'agent. Els resultats de les accions (resultat de fer les accions planificades) i els missatges són transmesos a la interfície del món

2.3.3.2. TouringMachines

TouringMachines és una arquitectura híbrida desenvolupada per Ferguson el 1992.

L'arquitectura (v. Figura 2.5) està formada per subsistemes de percepció i actuació els quals interactuen directament amb l'entorn de l'agent a través d'una interfície. Té un entorn de control (*framework control*) que està format per tres nivells. Els tres nivells són: reactiu (*reactive layer*), planificador (*planning layer*) i de modelat (*modelling layer*); cadascun d'aquests nivells és independent, executen processos concurrentment i poden comunicar-se entre ells amb pas de missatges.

El nivell reactiu (*reactive layer*) respon als events que succeeixen. Respon a events que provenen de l'entorn de l'agent (són percebuts), o que es requereixen dels altres nivells. Implementa un conjunt de regles de la forma situació-acció (amb pre/post-condicions).

El nivell de planificació (*planning layer*), construeix plans (seq. d'accions) per tal d'aconseguir algun objectiu de l'agent. Està format per dos elements: un planificador, i un mecanisme per focalitzar l'atenció (*focus of attention mechanism*). El planificador integra tant la generació com l'execució de plans; usa una llibreria de plans predissenyats. El *focus of attention mechanism* té la funció de filtrar la gran quantitat d'informació que el planificador ha de manejar, i així augmentar la seva eficiència.

El nivell de modelatge (*modelling layer*) conté una representació simbòlica de l'entorn de l'agent. Aquests models són manipulats per resoldre possibles conflictes entre objectius. Són situacions on la interacció d'objectius ha fet que un o més d'un, donat l'estat actual de l'entorn, serà impossible d'aconseguir per una interferència inesperada.

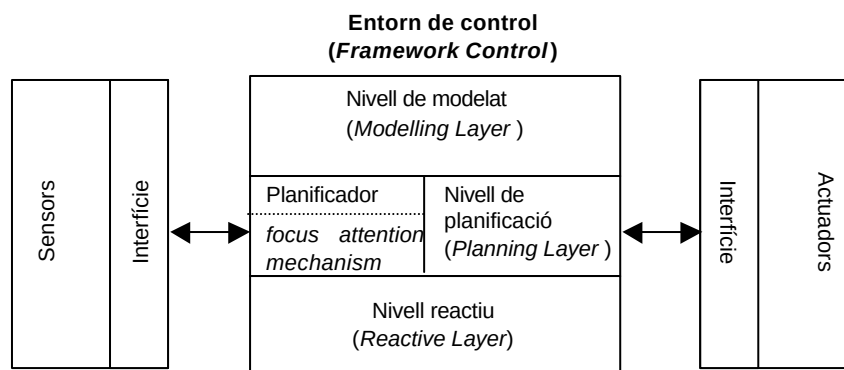


Figura 2.5: Arquitectura híbrida definida per TouringMachines.

2.4. Tipus d'agents

S'ha fet una possible classificació de diferents tipus d'agents. Són grans grups i potser una aplicació particular barreja alguna característica de més d'un.

Veurem llurs característiques fonamentals i les avantatges/inconvenients de cadascun.

TIPUS D'AGENTS	
Tipus bàsics	Col·laboratiu
	Interfície
	Mòbils
	Informació/Internet
	Reactius
	Híbrids
	Heterogenis

2.4.1. Agents d'informació/ Internet

Agents que tenen accés a les fonts d'informació (Internet) i poden posar en ordre i manipular informació obtinguda d'aquestes fonts, de manera que així poden respondre als requeriments imposats per l'usuari i/o altres agents. Molta gent es refereix als agents d'Internet com a agents que poden vagar per Internet per tal de recollir informació.

Avantatges i inconvenients

Cada cop hi ha més informació per xarxes electròniques i tenir un agent que proporcioni informació que l'usuari necessita, filtrant tot allò que no li interessa, pot ser de gran utilitat.

El principal problema d'aquests tipus d'agents és la gran quantitat d'informació que han de gestionar.

El repte més gran per a aquests tipus d'agents és crear una interfície d'usuari on la informació a buscar i aquella que es retorna sigui de fàcil entesa, fent-lo el més natural possible.

Exemples

JASPER [7] (Joint Access to Stored Pages with Easy Retrieval)	Treballa amb la xarxa i l'usuari (que li va donant consignes d'allò que vol), i és capaç d'emmagatzemar, recuperar, resumir i informar als altres agents/usuari de la informació útil que ha trobat a la WWW. Treballar directament amb la informació que hi ha en una URL és impracticable, per això Jasper condensa les dades d'una URL en meta-informació, que són les dates/temps de consulta, paraules clau de la plana i el títol d'aquesta. Cal a més, de manera local, fer una gestió de les pàgines que va captant, donat que ha de mantenir-les totes al dia. http://www.w3.org/Conferences/WWW4/Papers/180/
---	--

2.4.2. Agents d'interfície

Sistema on els agents col·laboren amb l'usuari per a resoldre un problema. L'agent observa i monitoritza accions preses per l'usuari, aprenent d'aquestes, i suggerint millors maneres de dur a terme certes tasques. L'aprenentatge es fa de quatre maneres diferents:

- Observant i imitant l'usuari.
- A través de rebre confirmacions positives o negatives de les accions.
- Rebent instruccions explícites de l'usuari.
- Demanant consell a altres agents.

Avantatges i inconvenients

L'agent pot adaptar-se a les preferències i hàbits de l'usuari.

La principal desavantatge és que aquests agents tendeixen a funcionar sols, o com a molt engeguen algun tipus de comunicació amb agents idèntics a ells. Podria ser més útil que fossin capaços de mantenir relacions amb qualsevol tipus d'agent d'interfície i cooperar en tasques [7].

Un inconvenient afegit és la crítica que fan alguns investigadors del tema, suggerint que no són necessaris uns agents que intentin aprendre del comportament de l'usuari, doncs l'usuari sap exactament allò que vol.

Exemples

LETIZIA	És un agent que assisteix en la navegació per la WWW. Proporciona una cerca cooperativa usuari-letizia. El seu paper és el de guia i va proposant pàgines a mesura que l'usuari va navegant; intenta endevinar el comportament futur i recomana nous destins. L'agent, dóna sempre les millors opcions utilitzant funcions heurístiques aplicades al comportament de l'usuari. http://lieber.www.media.mit.edu/people/lieber/Lieberary/Letizia/Letizia-Intro.html
---------	--

2.4.3. Agents de col·laboració

Aquests agents tenen com característiques principals la comunicació i la cooperació amb altres agents, per resoldre un problema en comú.

Per tal d'establir una bona col·laboració entre els agents, cal que negociïn.

La capacitat de resoldre problemes entre un grup d'agents és una tasca que s'ha de dur a terme amb tècniques d'IA distribuïda.

Totes aquestes tasques que han de dur a terme els agents col·laboratius tenen com a base fonamental la comunicació. Aquest fet és molt important en el desenvolupament d'entorns multiagent perquè cal una metodologia que permeti que s'entenguin entre ells (llenguatge de comunicació –ACL (*Agent Communication Language*)–) i un protocol de transmissió de dades (com per exemple TCP/IP).

En aquest tipus d'agents centrarem l'estudi del nostre projecte.

Avantatges i inconvenients

Avantatges:

- ➔ *Modularitat*. Permet descomposar el problema en subproblemes més simples.
- ➔ *Eficiència*. Deguda al paral·lelisme en la resolució de subproblemes.
- ➔ *Fiabilitat*. No hi ha una dependència d'un ordinador central.
- ➔ *Flexibilitat*. Noves tasques poden ser creades més fàcilment gràcies a la modularitat.
- ➔ *Reusabilitat*. Els agents poden distribuir el seu coneixement a més d'un agent.

Inconvenients:

- ➔ *Representació del coneixement*. Cal fer una traducció d'allò que és cert al món real a una notació que pugui ser tractada per l'agent. Normalment es fa servir lògica de primer ordre.
- ➔ *Coordinació entre els agents (inter-agent coordination)*. És essencial perquè els grups d'agents resolguin els problemes de forma efectiva.
- ➔ *Estabilitat i escalabilitat*. No és segur que els agents col·laborin sempre de forma estable. Tampoc no està resolt de forma efectiva el possible creixement en nombre d'agents en un sistema.
- ➔ *Aprenentatge*. L'aprenentatge és una funció que és interessant en aquests tipus d'agents, ara bé, el fet d'aprendre no és fàcil, i es pot qüestionar el fet que es tarda més temps aprenent que participant en un conjunt.
- ➔ *Avaluació/verificació*. Aquests, i en general tots el SMA, tenen la dificultat de ser verificats, és a dir, que compleixin les especificacions del problema.

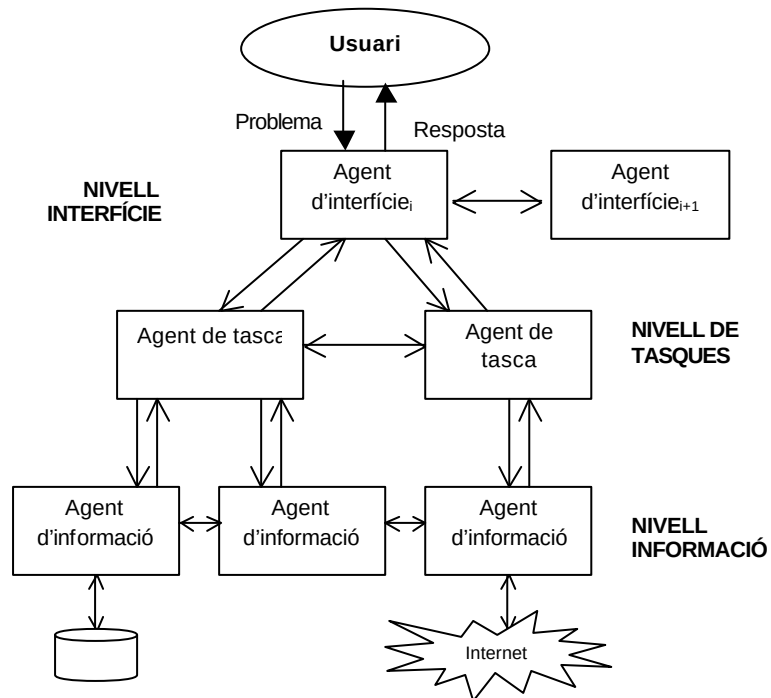
Exemples

PLEIADES

És una arquitectura distribuïda d'agents col·laboratius. Consta de tres nivells: el nivell més alt és la interfície amb l'usuari, un nivell de tasques específiques, i un nivell d'informació. L'usuari interactua amb l'agent d'interfície.

Dins d'aquest sistema es gestiona la petició de l'usuari, i aquests agents col·laboren entre ells. Per tal de captar informació requerida en aquest nivell, la demanen als agents d'informació al nivell d'informació. Aquests a la vegada col·laboren entre ells (p.e. per no repetir informació, per intercanviar-ne, etc). La font d'informació són les BBDD d'un entorn específic, o la www, o dades extretes d'emails.

Al final l'agent de tasca proposa una solució al nivell d'interfície. Aquest nivell recopila tota la informació del nivell de tasques, i dona una proposta resultant a l'usuari, que en última instància pot acceptar o no.



<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-5/www/pleiades.htm>
<http://www.labs.bt.com/projects/agents/publish/papers/review3.htm>

2.4.4. Mòbils

Aquests agents tenen com a principal característica la de poder moure's per una xarxa electrònica (per exemple WWW). Aquesta mobilitat es fa interactuant amb altres hosts, i permet als agents recollir informació, així com realitzar serveis requerits per l'usuari.

Els agents mòbils col·laboren i tenen autonomia, però són diferents als agents col·laboratius.

Avantatges i inconvenients

Avantatges:

- ➔ *Cost de comunicació reduït.* La mobilitat dels processos fa que s'executin localment i es redueixi la transferència que es faria en el cas de client-servidor.
- ➔ *Fonts locals limitades.* El poder de processat i la capacitat d'emmagatzematge en una màquina local pot ser limitada, per això és necessari l'ús d'un agent mòbil.
- ➔ *Computació asíncrona.* Es pot fer funcionar els agents mòbils i fer alguna tasca; els resultats aniran cap al mailbox. Poden operar quan, fins i tot no estem connectats.
- ➔ *Arquitectura de computació distribuïda flexible.* Proporcionen una nova manera de fer computació distribuïda.

Inconvenients:

- ➔ *Transport.* Cal trobar una manera de poder moure els agents d'un lloc a l'altre.
- ➔ *Autenticació.* Cal autenticar que un agent sigui allò que diu que representa.
- ➔ *Secret.* Cal que els agents mantinguin la privacitat de l'usuari. Cal a més, que ningú pugui accedir al nostre agent i que l'executi sense nosaltres saber-ho.
- ➔ *Seguretat.* Cal protegir/prevenir de virus.
- ➔ *Directoris.* Cal saber els serveis/llenguatge d'un agent per tal de fer-lo servir.

Exemples

Telescript	Telescript és llenguatge de programació orientat a objectes que permet d'implementar models de programació remota. Permet de fer "viatjar" a través d'una WAN (Internet) programes que són transmesos de manera <i>segura</i> (les dades s'empaqueten per la transmissió afegint informació de verificació) . La forma bàsica de treballar de Telescript és: 1) el propietari fa "anar" l'agent cap al destinatari, 2) aquest negocia els serveis, 3) els usa, 4) paga i retorna al propietari. http://science.gmu.edu/~mchacko/Telescript/docs/telescript.html
------------	--

2.4.5. Agents reactius

Els agents reactius tenen com a particularitat que no treballen amb models simbòlics de l'entorn. Enlloc d'això, actuen/responen a un estímul, segons un patró de l'estat actual en el qual es troben. El fet d'actuar segons un estímul seguint un patró els fa realment simples; també ho són les seves interaccions amb altres agents.

Les tècniques, arquitectures i d'altres teories que s'han formulat sobre aquests tipus d'agents (Brooks, 1986), s'han desenvolupat perquè aquesta simplicitat no és tant obvia quan es tenen sistemes que estan formats de nombrosos agents. Per exemple, Brooks proposa una arquitectura modularitzada, on cada mòdul és independent (reactiu) però alhora aquests mòduls estan pensats perquè cadascun estigui controlat per un altre, formant una jerarquia.

Aquests agents, segons Maes, tenen tres característiques principals:

- 1) La capacitat del grup es veu augmentada a la d'un element sol.
- 2) Podem descomposar un problema en tasques que poden ser resoltes pels diferents mòduls independentment,
- 3) Els agents reactius treballen directament sobre les dades dels sensors, i no treballen sobre alts nivells d'abstracció com seria treballar sobre símbols.

Avantatges i inconvenients

Avantatges:

➔ *Simplicitat i fàcil enteniment.* Aquests tipus d'agents no contenen un raonament abstracte i per això s'entenen amb facilitat. El seu funcionament permet rapidesa perquè han de "recordar" poc. Les seves accions depenen només d'allò que passa al moment present.

➔ *Flexibilitat.*

➔ *Adaptabilitat.*

➔ *Resposta ràpida i no fragilitat.*

➔ *Resolució dels problemes de frame (estructura).* Aquests problemes són intractables amb les tècniques d'IA tradicionals.

Inconvenients:

➔ *Poques aplicacions.* Actualment només es troben aplicacions limitades, principalment en jocs i simulacions. És per això que hi ha una clara necessitat d'expandir el rang de llenguatges, teories, arquitectures i aplicacions per als agents reactius.

➔ *Dificultat d'adaptació a un nou entorn.* Es desconeix què passa quan es canvia l'entorn i com es corregeixen/detecten els possibles errors

➔ *Manca d'objectius explícits.*

2.4.6. Híbrids

Aquests tipus d'agents pretenen juntar diferents tipus d'agents per tal de reunir-los en un sol agent, de tal forma que es minimitzin els desavantatges de cadascun d'ells per separat. Hom pot pensar en un agent híbrid que estigui format per una part deliberativa i una part reactiva, així els processos que hagin de ser ràpids i previstos a priori poden tractar-se amb la part reactiva, i les accions que comportin una planificació es realitzen a la part deliberativa.

Avantatges i inconvenients

Avantatges:

En aplicacions concretes, la combinació de diferents agents proporciona més avantatges que no pas els guanys obtinguts quan només tenim un agent sol. Per tant, idealment, es tracta de maximitzar els beneficis obtinguts dels agents individuals en un conjunt (híbrid). Segons quin tipus d'agents estiguin formant part de l'híbrid, tindrem diferents característiques. Per exemple, si tenim el cas d'una part reactiva i una de deliberativa, i volem que tingui més prioritat la part reactiva, es produiran els següents efectes:

- *Robustesa.*
- *Resposta ràpida.*
- *Adaptabilitat.*
- *Resolució dels problemes de frame.*

Si per contra, volem que tingui més importància la part deliberativa de l'agent, hi haurà una millor orientació als seus objectius.

Inconvenients:

- *Poques architectures basades en agents híbrids.* El seu efecte però és força contundent.
- *Disseny sense principis.*
- *Moltes architectures híbrides tendeixen a tenir aplicacions massa específiques.*
- *Usualment les teories que sostenen els sistemes híbrids tampoc estan especificades.*

Exemples

INTERRAP	Interrap no només és un tipus d'agent, a més defineix una arquitectura d'agents. Intenta agafar allò que és millor d'alguns tipus d'agents i els junta per unir les principals avantatges, i alhora reduir els desavantatges que poden tenir cadascun per separat. Interrap junta una part deliberativa i una reactiva (v.apartat Arquitectures híbrides). http://www.dfki.uni-sb.de/~pischel/interrap.html
----------	--

2.4.7. Heterogenis

A diferència dels agents híbrids, els sistemes heterogenis es refereixen a un conjunt integrat d'almenys dos o més agents que pertanyen a dues o més classes diferents d'agents. Pot, doncs, contenir un o més agents híbrids.

L'argument de la seva existència es basa en què el món és abundant en una rica diversitat de productes de software, que proporcionen un ampli rang de serveis. Encara que els programes treballin aïllats, hi ha una forta demanda per tal que aquests 'inter-operin'. El requeriment per a aquesta inter-operació és que hi hagi un llenguatge de comunicació per a agents, per tal que els diferents agents es puguin comunicar.

Tenim agents quan i només si es comuniquen correctament en un llenguatge de comunicació per a agents. Si aquests agents, ara es col·loquen junts en un conjunt heterogeni, també és possible tenir un nou agent.

Per a ser un software hereditat, necessiten primer ser convertits en un agent de software (*agentificació*). Hi ha tres formes d'agentificar:

- a) *Traductor*. Aquest ha de poder passar el llenguatge de comunicació d'agents al protocol d'entrada del programa; i del tipus de sortida del programa al llenguatge de comunicació. Encara que el programa no cal modificar-lo, en traduir, es perd eficiència.
- b) *Embolcar (Wrapper)*. S'afegeix directament al codi del programa el necessari per a poder llegir/escriure missatges en el llenguatge de comunicació dels agents. Tot i que aquesta solució és més eficient, cal modificar cada programa en concret.
- c) *Re-escriure*. Solució dràstica on re-escrivim en un nou llenguatge el programa vell. Podem optimitzar al màxim les comunicacions però, cal fer la feina de nou.

Avantatges i inconvenients

Avantatges:

→ *Nombre elevat d'aplicacions*.

→ *Possibilitat d'embolcar programes anteriors*. La tecnologia d'agents heterogenis pot protegir o reduir l'efecte del manteniment de rutina del software, implementar millores, etc.

→ *Disseny de software*. L'enginyeria de software basada en agents proporciona una nova aproximació al disseny, implementació i manteniment en general, així com a la interoperabilitat en particular.

Sovint es compara aquest tipus de software basada en agents a la programació orientada a un objecte, proporcionant una interfície amb missatges basats en dades estructurals internes i algorismes. Ara, cal utilitzar un llenguatge comú amb semàntica independent dels agents.

Es solen plantejar tres qüestions sobre ells que podrien representar-ne els inconvenients:

- Quin és el llenguatge de comunicació per a agents més apropiat?
- Com són capaços els agents de comunicar-se amb aquest llenguatge construït?
- Quines arquitectures de comunicació són favorables per a la cooperació?

Exemples

PACT (<i>Palo Alto Collaborative Testbed</i>)	Aproximació d'un SMA integrat per quatre sistemes i formant un esquelet comú. Inclou 31 programes basats en agents, treballant en 15 estacions de treball i microcomputadors. Els agents estan embolcallats amb <i>agents d'informació</i> els quals fan de pont entre el sistema de coneixement i les funcions que realitzen. [11] pags. 30-32 per informació.
---	---

3. SISTEMES MULTIAGENT

Com s'han vist, els agents no són tots iguals i tenen característiques diferents. Els agents individualment proporcionen una funcionalitat molt interessant, però el que els dona un valor afegit és la capacitat d'*unir-se cooperant* per a resoldre problemes que sols potser no podrien solucionar o els hi seria molt difícil. A més, normalment el coneixement que té cada agent és limitat, pot ser incorrecte o incomplet, i per això cal que formi part d'un col·lectiu al qual hi participa i del qual se'n beneficia.

Aquesta és la finalitat dels Sistemes Multiagent (SMA)³ (*Multiagent Systems*), i la base d'aquesta participació/benefici és la *comunicació*.

3.1. Comunicació

Si cada agent tingués la informació completa sobre l'estat del sistema, llavors la cooperació podria ser processada i programada a priori, sense cap tipus de comunicació. Els agents podrien treballar junts com un equip perfectament coordinat, sabent tothom que hauria de fer. Ara bé, excepte en els sistemes més trivials, aquest perfecte coneixement és impossible d'assolir. Els agents, en la realitat, tenen informació parcial, possiblement incorrecta sobre l'estat del seu entorn. Per això, per tal de cooperar efectivament els agents s'han de comunicar els uns amb els altres i han de treballar amb algun llenguatge de comunicació per agents (*Agent Communication Language (ACL)*)⁴.

LLENGUATGES DE COMUNICACIÓ PER AGENTS (ACL)

Els ACL han de respondre a totes les possibles necessitats que els agents puguin requerir en qualsevol tipus de comunicació. Els agents necessiten comunicar informació, intencions, objectius, i tot allò relacionat amb l'entorn.

Els ACL estan basats en la teoria de la parla (*speech act theory*), en la qual, les comunicacions individuals entre agents poden reduir-se a un petit nombre d'idees o més generalment actes comunicatius. Aquests actes donen forma al significat bàsic de la comunicació.

L'aspecte d'un missatge d'un ACL és de la següent manera:

```
accept [to: agent_2, from: agent_3, reference: contract_4, cost: £40]
```

On especifiquem que l'agent_3 diu a l'agent_2 que ha acceptat el contract_4, que té un cost de £40.

Actualment, no hi ha cap ACL àmpliament usat i podem dividir els diferents tipus existents en dos: *estàndard* i *ad hoc* [7].

³ Farem servir la sigla SMA fent referència a Sistema Multiagent.

⁴ D'ara en endavant, la sigla ACL farà referència a Llenguatge de Comunicació entre Agents.

3.2. Llenguatges de comunicació (ACL) estàndards

Anteriorment hem dit que no hi ha un llenguatge que actualment s'hagi imposat com estàndard en el desenvolupament d'entorns multiagent. Actualment s'han desenvolupat diversos llenguatges, sense cap que s'imposi àmpliament. Els més utilitzats per la comunitat científica són l'ACL de la *Foundation for Intelligent Physical Agents* (FIPA), i el *Knowledge Query and Manipulation Language* (KQML).

3.2.1. Llenguatge de comunicació de la FIPA

Hi ha una organització internacional que està dirigint les pautes que han de seguir els ACLs. Aquesta organització és la *Foundation for Intelligent Physical Agents* (FIPA). En [5] es pot trobar tota la informació necessària pel desenvolupament d'un ACL⁵ actualitzada periòdicament.

Aquesta organització defineix l'estructura i funcionament d'un ACL de manera modular. Defineix com s'han de comunicar els agents, el format del missatges, el contingut d'aquests, etc. (v. Figura 3.1), tot allò necessari per realitzar totes les funcions que poguéssim pensar: cooperació, negociació, mobilitat d'agents, transport de dades/objectes, etc.

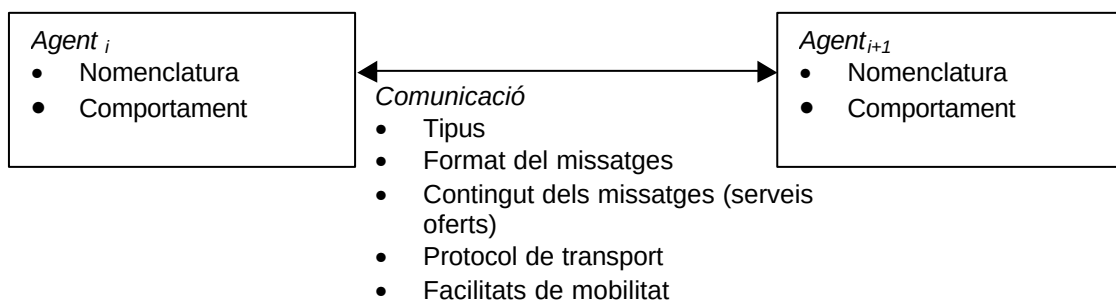


Figura 3.1: Domini de la definició FIPA

L'estàndard FIPA defineix que tot SMA amb la seva especificació ha de tenir un *Agent Directory* que conté tota la informació sobre els agents, descripció de l'entorn i serveis que poden ser accedides pels agents.

⁵ Al llenguatge de comunicació estàndard de la FIPA (*Foundation for Intelligent Physical Agents*), ens hi referirem per FIPA ACL

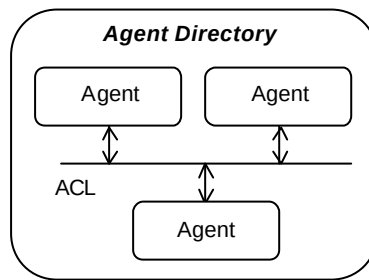


Figura 3.2: Elements d'un domini SMA.

En sistemes federats (SMA que està format de dos o més dominis diferents), cada domini ha d'implementar un *Domain Facilitator* per tal de facilitar la comunicació entre els diferents dominis.

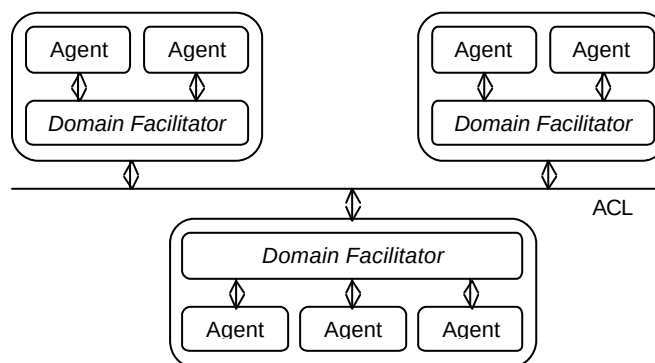


Figura 3.3: Sistema federat amb llurs interconnexions entre els diferents dominis.

En la Figura 3.3 es tenen tres màquines amb dos o tres agents cadascuna. Els agents no es comuniquen directament entre ells, sinó que es comuniquen a través de *facilitadors*.

La funció principal dels facilitadors és la de mitjancers, els quals són capaços de localitzar altres agents en la xarxa i són capaços de proporcionar altres serveis (traducció de llenguatges). També estableixen la connexió a través de l'entorn per permetre les “converses” entre agents. Aquestes converses inclouen tècniques de cooperació/negociació per resoldre tasques.

Els agents no els hi cal parlar el mateix llenguatge, poden treballar en Lisp, Java, etc., i és el facilitador qui tradueix a un ACL comú que es transmet.

FIPA especifica tot aquest entorn.

Exemples

FIPA_SMART	Eina de desenvolupament de SMA que implementa les recomanacions realitzades per la FIPA. http://dirac.nosc.mil:6996/FIPA_SMART/FipaSmartIntro.html
------------	--

3.2.2. Knowledge Query and Manipulation Language (KQML)

Knowledge Query and Manipulation Language (KQML) és un llenguatge i protocol per intercanviar informació i coneixement.

Aquest treball va ser realitzat formant part d'altres projectes desenvolupats per l'organització *ARPA Knowledge Sharing Effort*⁶. Aquesta entitat té la funció de dissenyar tècniques i metodologies per construir a gran escala bases de coneixement que puguin ser distribuïdes i reutilitzables.

KQML defineix un conjunt extensible de *performatives* o tipus de funcions, les quals defineixen el conjunt permès d'actes de comunicació (*speech act*) que els agents poden utilitzar.

La sintaxi de KQML és similar a la de Lisp (podria fàcilment implementar-se KQML usant Lisp). Amb algunes paraules clau i unes regles de puntuació, els missatges KQML són fàcils de llegir pels humans.

KQML i FIPA poden ser vist modularment. Concretament KQML es pot veure en tres nivells: *comunicació*, *missatge* i *contingut* [7, 12] (v. Figura 3.4).

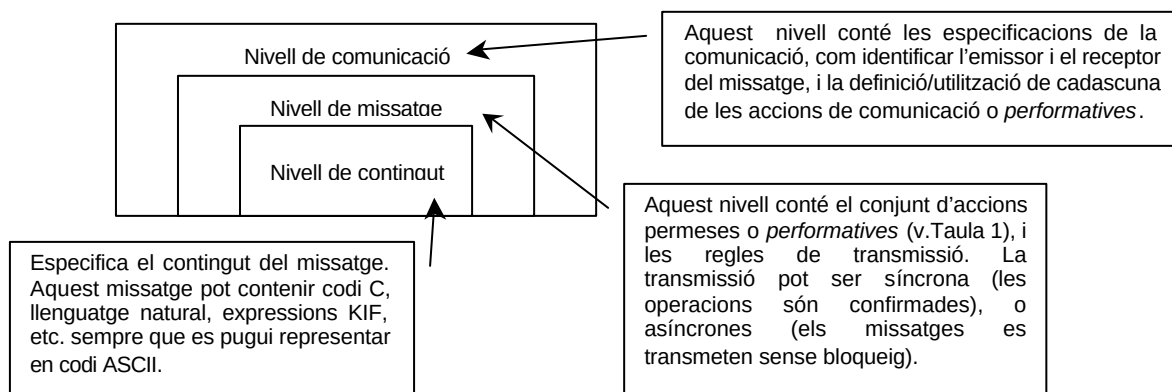


Figura 3.4: Nivells d'abstracció de l'ACL KQML.

⁶ *ARPA Knowledge Sharing Effort (KSE)* és un consorci que desenvolupa convencions que faciliten la distribució i reutilització de bases de coneixement i sistemes basats en el coneixement. Els objectius d'aquest consorci són definir, desenvolupar i testear infraestructures i tecnologies, per permetre construir sistemes distribuïts molt més potents i amb rendiments molt més alts, que sistemes locals o aïllats. El KSE està organitzat en quatre grups de treball: *Interlingua Group*, *Knowledge Representation System Specification Group (KRSS)*, *Share, Reusable Knowledge Bases Group (SRKB)* i *External Interfaces Group*. El primer, desenvolupa llenguatges per expressar els continguts d'una base de coneixement; aquest grup va desenvolupar *Knowledge Interchange Formalism (KIF)* (v. apartat següent d'aquest document). El segon grup, *KRSS*, defineix les estructures de representació. El tercer grup, *SRKB*, intenta definir com s'ha de realitzar la distribució del coneixement, donant eines i metodologies. L'últim grup, l'*External Interfaces Group*, estudia les interaccions entre sistemes basats en el coneixement i altres mòduls del seu entorn, principalment entre dos sistemes basats en el coneixement, i un sistema basat en el coneixement i una base de dades; un dels resultats d'aquest grup de treball és la definició de *Knowledge Query and Manipulation Language (KQML)*. [12]

Encara que es té un conjunt reduït de *performatives* o accions permeses pels agents, KQML suporta diferents tipus d'arquitectures. Les *performatives* descriuen meta-dades que especifiquen requeriments i capacitats dels agents. En aquest intercanvi de coneixement es va introduir la figura d'un tipus especial d'agent: *communication facilitator*. Aquest agent pot desenvolupar diverses funcions de comunicació, com per exemple, mantenir un registre dels serveis, encaminar missatges, intercedir entre proveïdors i clients d'informació, traduir serveis, etc.

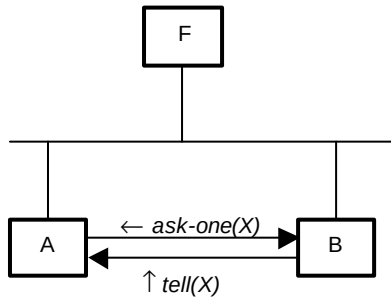
El cos de KQML té només trenta sis tipus d'accions (*performatives*) agrupades en nou categories (v. Taula 3.1), que defineixen les possibles accions de comunicació que els agents poden utilitzar, i les quals proporcionen el substrat per a realitzar coordinació i estratègies de negociació entre ells [11].

Tipus d'acció (performative)	Paraules KQML reservades
Accions bàsiques d'informació	<i>tell, deny, untell, cancel</i>
Accions bàsiques de requeriment	<i>evaluate, reply, ask-if, ask-about, ask-all, ask-one, sorry</i>
Accions de requeriment multi-resposta	<i>stream-about, stream-all</i>
Accions de confirmació	<i>achieve, unachieve</i>
Accions de generació	<i>standby, ready, next, rest, discard, generator</i>
Accions de definició de capacitats	<i>advertise</i>
Accions de notificació	<i>subscribe, monitor</i>
Accions de comunicació	<i>register, unregister, forward, broadcast, pipe, break</i>
Facilitats	<i>broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all</i>

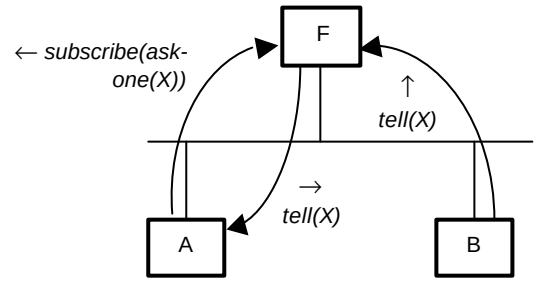
Taula 3.1: Tipus de missatges o performatives que poden transmetre's en KQML

En la Figura 3.5, podem veure alguns exemples d'utilització del llenguatge de comunicació entre agents KQML.

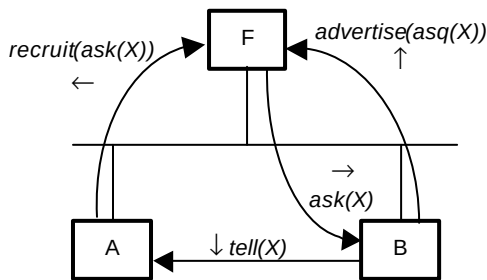
Les possibilitats de serveis proveïts per les diferents performatives cobreixen molts tipus de necessitats.



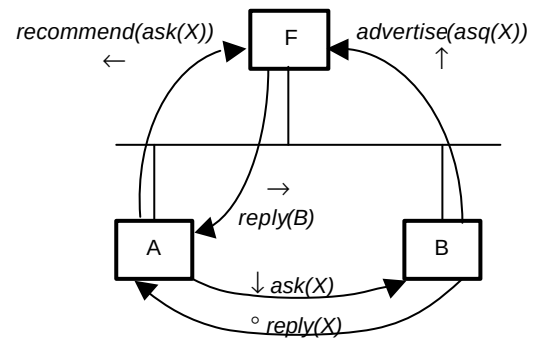
a) Si l'agent A és conscient de les característiques/funcions i la localització de l'agent B, pot enviar-li qualsevol requeriment directament. És una comunicació punt a punt.



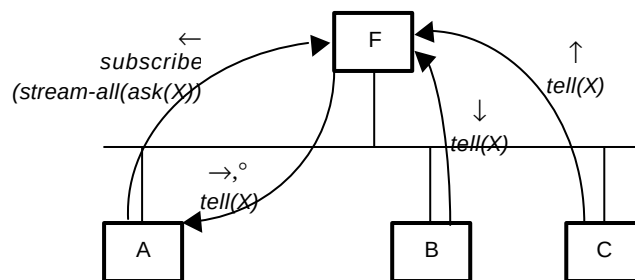
b) L'agent A demana a l'agent facilitador (F) que li monitoritzi els canvis en la seva base de coneixement. L'agent A es subscriu a l'agent F i demana que li sigui enviada qualsevol informació referida a la seva *performative* enviada. En aquest cas l'agent A només demana el primer canvi.



c) *Recruit* "recluta" agents per a realitzar tasques determinades. L'agent A busca agents que li puguin avançar algun tipus de *performative*. És a dir, si l'agent B pot satisfer la *performative* requerida per l'agent A, l'agent B envia directament la informació a l'agent A sense passar per l'agent F.



d) La comanda *recommend* es fa servir per preguntar a l'agent facilitador (F) el nom d'un altre agent que sigui indicat per fer alguna tasca. Els agents prèviament han indicat a F les seves característiques/possibilitats amb un *advertise*.



e) En aquesta configuració l'agent A dona a l'agent F la missió de *monitoritzar* els canvis que es produeixen sobre X; va rebent del facilitador les evolucions sobre X. La combinació *subscribe/stream-all* pot simplificar-se amb la comanda *monitor*.

Figura 3.5: Exemples d'utilització de diferents tipus d'accions de KQML o *performatives*.

Els tipus de missatges KQML o *performatives*, tenen una estructura fixa que conté diferents camps (v. Taula 3.2). Ara bé, depenent del tipus de missatge que es vulgui transmetre, hi ha paràmetres del missatge que es poden obviar, o no. Cal veure l'especificació que es dóna per cadascuna de les accions.

Paràmetre	Significat
:content <expression>	Indica que <i>expression</i> (que pot ser un caràcter, una paraula, una cadena, o una successió d'aquests elements), és el contingut del missatge.
:force <word>	Si <i>word</i> és permanent, qui envia el missatge garanteix que mai refusarà els missatges. Qualsevol altre valor indica que el remitent del missatge pot refusar missatges en un futur. Aquest paràmetre permet d'evitar missatges innecessaris.
:in-reply-to <expression>	<i>expression</i> és l'identificació del receptor en cas de resposta.
:language <word>	<i>word</i> és el nom del llenguatge del contingut del missatge (:content).
:ontology <word>	És el nom de l'ontologia usada en el paràmetre :content. En aquest cas :content ha de pertanyer a aquesta ontologia, o si no hi és a l'ontologia per defecte.
:receiver <word>	Indica el receptor del missatge o <i>performative</i> .
:reply-with <expression>	Si <i>expression</i> és t, qui envia el missatge espera una resposta, sinó no.

Taula 3.2: Paràmetres dels missatges en KQML.

Com s'ha vist, KQML permet l'intercanvi de coneixement i dades entre agents permetent que aquesta informació estigui en diferents llenguatges. Si volem ajuntar dues o més aplicacions que treballen en llenguatges diferents, cal afegir traductors dels diferents llenguatges.

Per solucionar aquest problema, l'*Interlingua Group* del consorci ARPA KSE, va formalitzar un llenguatge per expressar el contingut d'una base de coneixement. Aquest llenguatge és *Knowledge Interchange Formalism* (KIF).

3.2.2.1. Knowledge Interchange Format (KIF)

Knowledge Interchange Format (KIF) és un llenguatge dissenyat per ser usat en l'intercanvi de coneixement entre sistemes de computadors.

KIF no està pensat com un llenguatge amb el qual un humà pot interactuar (encara que pot ser usat per aquest propòsit). Per interactuar-hi calen eines pensades per això: grafs conceptuals, llenguatge natural, llenguatges com Prolog, i d'altres interfícies.

El coneixement no es guarda en la notació KIF, és a dir, quan un computador llegeix el coneixement en base KIF, aquest converteix la informació en estructures de dades en la seva pròpia forma (estructures de punters, matrius, vectors, ...). Llavors la computació es fa usant aquestes estructures de dades. Quan un computador necessita comunicar-se amb algú altre, cal fer una conversió d'aquestes estructures internes a notació KIF.

Característiques principals:

- ➔ *Semàntica declarativa*. Aquest fet fa possible entendre les expressions KIF sense utilitzar cap intèrpret que les manipuli.
- ➔ *És comprensible ògicament*. No només proporciona operacions lògiques bàsiques, sinó que es poden definir de noves arbitràriament.
- ➔ *La representació del coneixement es fa sobre coneixement*. És a dir, l'usuari pot crear noves decisions sobre el coneixement explícitament, o introduir nou coneixement directament, sense canviar el llenguatge.
- ➔ *Implementabilitat*. Encara que KIF no està pensat per ser utilitzat dins de programes, com a representació/intercanvi de coneixement pot ser utilitzat/implementat.
- ➔ *Simplicitat*. Un humà pot entendre'l fàcilment.

Sintàcticament i conceptualment, aquest llenguatge és molt semblant a Lisp.

Els objectes bàsics de KIF són llistes (formades per nombres, caràcters, paraules, constants, objectes definits per l'usuari, etc.) i llurs relacions (relacions lògiques, funcions, etc.). Igual que Lisp, es fa servir notació prefix (v.Taula 3.3).

Expressió KIF	Explicació
(if (> 1 2) 1 (> 2 1) 2 0)	és equivalent a 2.
(if (> a 0) a (- a))	equival al valor absolut d'a.
(defobject s := t) (= s t)	Definim l'objecte s amb contingut del paràmetre t.
(deffunction f (v1 ...vn) := t) (= (f v1 ...vn) t)	Mirem si els elements d'una llista són tots iguals.
(defrelation even (?x) := (integer (/ ?x 2))	Mirem si un n ^o és o no parell
(listof mary (listof tom dick harry) sally)	Creem la llista (mary tom dick harry sally)
(deffunction remove (?x ?l) := (cond ((null ?l) nil) ((and (= ?x (first ?l)) (list ?l)) (remove ?x (rest ?l))) ((list ?l) (cons ?x (remove ?x (rest ?l))))))	Borrem l'element x de la llista l.

Taula 3.3: Exemples de notació KIF (exemples extrets de l'especificació de KIF [8])

3.3. Llenguatges de comunicació (ACL) *ad hoc*

Per algunes aplicacions els llenguatges estàndards potser no són massa eficients o no responen a les especificacions de les transmissions. Per això, es poden crear conjunts d'accions que, definides com KQML, responguin a allò que necessitem.

Per exemple, *SRI Open Agent Architecture* defineix un ACL anomenat *Inter-agent Communication Language* (ICL), que només defineix tres tipus de missatges: *solve*, *do* i *post*.

Aquests llenguatges fets a mida tenen el defecte de no poder-se estendre a sistemes ja fets. Si fem servir un llenguatge estàndard, poden pensar en ajuntar o fer cooperar diferents sistemes ja implementats

Exemples

AgentTalk	<i>AgentTalk</i> defineix un protocol per facilitar la coordinació en sistemes multiagent. Està definit el Lisp, i permet la implementació de tècniques de cooperació/negociació, com el <i>contract net</i> i altres. Poden afegir-se més funcions. Ha estat creat per NTT Communication Science Laboratories i Ishida Laboratory, però malauradament, el software només es distribueix al Japó.
Bee-gent	<i>Bee-gent</i> defineix un ACL basat en KQML, però usant un nou protocol de comunicació a través d'Internet, l'XML que l'anomena XML/ACL. Defineix un entorn pensat per realitzar transaccions amb negociació i <i>agent wrappers</i> per aplicacions ja realitzades. http://www.toshiba.co.jp http://www2.toshiba.co.jp/beegent/tutorial/tindex.htm

3.4. Negociació i cooperació en SMA

Els agents *habiten* en un entorn distribuït i han de coordinar les seves activitats. La cooperació i la negociació incrementen les possibilitats/funcions dels agents individualment, i del conjunt del sistema. Per aconseguir aquests propòsits es poden aplicar diferents tècniques o estratègies que els fan possibles.

Hi ha alguns paràmetres que cal tenir en compte a l'hora d'escollir la tècnica de coordinació adequada [13]:

- ***El nivell de cooperació entre els agent.*** Cal veure en quin cas són: en un entorn on tots els agents treballen per ells mateixos, en un entorn on tots els agents tenen un mateix objectiu i treballen conjuntament per aconseguir-ho, o en un altre cas intermediari.
- ***Relacions entre els agents i protocols.*** Cal avaluar les situacions que es poden establir entre els agents, i cal mirar si es poden definir protocols o estratègies a priori, o si les diferents interaccions no poden pensar-se a priori i donar aquesta responsabilitat als agents.
- ***Número d'agents.*** La coordinació s'estableix a través de missatges. Segons l'estratègia utilitzada, si el número d'agents és gran (centenars), pot no ser factible la seva implementació.
- ***Tipus d'agents.*** Els sistemes poden estar formats només per agents amb un comportament predefinit, o podem tenir sistemes i usuaris, on hi ha molta més indefinició.
- ***Comunicació i cost de computació.*** Cal avaluar tots els elements que involucren els SMA. Com processos que es comuniquen, cal tenir en compte la quantitat d'informació que es transmet i el seu cost.

Hi ha diverses tècniques per implementar aquestes estratègies de comunicació:

- ***Teoria de Jocs***
- ***Contract Net Protocol***
- ***Distributed Problem Solve***

3.4.1. Teoria de Jocs

Aplicar tècniques de *Teoria de Jocs*. La Teoria de Jocs estudia models matemàtics basats en interaccions i cooperacions entre gent. Els models d'aquesta teoria són representacions de situacions del món real que involucren els diferents individus (jugadors), els quals tenen diferents objectius i preferències.

Es poden tenir dos tipus diferents de models: els no cooperatius, on els jugadors funcionen per si sols, o cooperatius, on les accions són preses per grups de jugadors.

La utilització d'aquestes tècniques requereixen gran quantitat de càlculs i comunicacions.

3.4.2. Contract Net Protocol (CNP)

Una de les eines més exteses de coordinació és el *contract net protocol* (Smith, 1980).

El funcionament és senzill:

El *manager* envia una petició d'oferiment (especificant una acció que ha de ser satisfeta, un criteri per avaluar l'oferta, i un *timeout* o caducitat del missatge) a un conjunt de *possibles interessats*. Els agents d'aquest grup *refinen* la proposta (segons les seves necessitats/possibilitats) i la transmeten al *manager* que l'ha iniciada. Si no hi ha ningú interessat, la cooperació falla.

El *manager* ordena les propostes segons un criteri propi, i es tindran tres tipus de casos: els agents que no han respost en el *timeout*, l'agent que ha respost i proposa la millor oferta, i la resta d'agents.

El *manager* accepta les condicions del millor postor i li envia allò que ha ofertat.

Cal veure que la coordinació encara pot fallar perquè pot no executar-se correctament la tasca al destí.

El funcionament bàsic del *contract net* pot optimitzar-se evitant comunicacions innecessàries.

3.4.3. Distributed Problem Solve (DPS)

Els agents autònoms treballen en entorns DPS que pot considerar-se una organització, i en aquests casos es poden aplicar algorismes de DAI (*Distributed Artificial Intelligence*).

Aquest algorisme funciona en entorns on hi ha relativament pocs agents (com a molt, una dotzena) perquè el cost computacional de les operacions és normalment alt.

Donat un conjunt d'agents i un conjunt de tasques que s'han de satisfer, es considera que cada tasca és realitzada pel grup d'agents, no per cap en especial. És la distribució de la feina i després ensamblar els subresultats.

4. APLICACIONS DELS SISTEMES MULTIAGENT

Els sistemes multiagent tenen un ampli ventall d'aplicacions. Actualment, la majoria es troben en fase de desenvolupament.

Els àmbits d'aplicació són:

Assistents personals. Aquests sistemes treballen amb i per a l'usuari. Es comuniquen amb l'usuari per ajudar-lo a dirigir activitats diàries, email, agenda personal, etc. Poden comunicar-se amb altres agents per recollir informació. Actuen semiautomàticament i intenten aprendre d'allò que fan per tal d'avançar-se a l'usuari si fos possible (iniciativa). Per exemple podem trobar sistemes d'aprenentatge de perfils d'usuari, sistemes d'interfície multimedia, o assistents personals digitals.

Aplicacions de recuperació d'informació. Aquests sistemes incorporen tots aquells serveis que l'usuari necessita per trobar informació fàcil i ràpidament. Podem trobar serveis de directori (pàgines grogues), cerca d'informació en bases de dades, etc.

Aplicacions d'entreteniment. Aquests sistemes inclouen:

- Jocs (interacció entre personatges autònoms i amb l'entorn i, jocs multi-usuari).
- Aplicacions destinades a arcades, WWW i màquines de jocs.
- Producció de pel·lícules/vídeo: a) càmeres (amb moviment automàtic, focus, etc.), b) gràfics 3D i films animats, dibuixos animats, etc.

Aplicacions en gestió de serveis. Aquests són sistemes que inclouen la configuració i distribució dels serveis requerits per l'usuari. Entre aquests destaquen:

- Serveis multimèdia
- Serveis compra/venda.
- Serveis de gestió de xarxes intel·ligents
- Guies de serveis i de viatges

Aplicació en gestió de negocis. Aquests sistemes fan front o resolen problemes de gestió de negocis i tasques de proporcionar recursos. També per a dur a terme operacions financeres. Inclou:

- Comerç electrònic
- Serveis financers.
- Automatització d'oficines.
- Telecomunicació.
- Organització d'empreses.

Aplicació en gestió de la manufactura. Aquests sistemes inclouen agents designats a tractar amb la gestió de tasques i processos de l'entorn industrial. Aquests processos poden prendre part en el control de robots industrials i màquines a través d'interfícies. Algunes aplicacions són:

- Gestió de robots.
- Automatització d'una factoria (cadena de muntatge).
- Balancejat de càrrega.

Aplicacions en robots. Més específicament que la gestió de robots industrials, aquestes aplicacions estan pensades per a robots domèstics o d'oficines no estructurades. Per exemple podem pensar en robots de neteja, o repartiment de correu en una oficina.

Aplicacions de recerca. Altres àmbits en els quals es continua investigant l'aplicació d'agents són:

- Tractament d'imatges.
- Sistemes d'aprenentatge i adaptació.
- Processat del llenguatge parlat i de la veu.
- Sistemes de coneixement distribuït.
- Interfícies home-computadora.

5. LLISTA D'EINES DE DESENVOLUPAMENT SMA

Producte	URL	Companyia	Llenguatge	Descripció
Agent Building Shell (ABS)	http://www.ie.utoronto.ca/EIL/ABS-page/ABS-overview.html	University of Toronto	COOrdination Language (COOL)	Arquitectura d'agents
Agent Development Environment (ADE)	http://www.gensym.com/products/g2intelligentagents.htm	Gensym		G2 Agent Development Environment (ADE) s'utilitza per crear sistemes distribuïts intel·ligents basats en tècniques d'agents. És un producte registrat.
Agent Development Tools	http://www.ai.sri.com/~ooa/adt.htm	SRI International	Suporta diferents llenguatges (Prolog, C, Delphi, Visual Basic, Lisp)	SRI proporciona un conjunt d'eines per desenvolupar agents (<i>Agent Development Tools</i> (ADT)). El sistema consta de tres eines: PROACT (<i>Programmers Agent Construction Tool</i>) per crear el sistema multiagent i llurs funcions; LEAP (<i>Linguistic Expertise Acquisition Program</i>) és un programa que dóna possibilitat als agents de comunicar-se usant llenguatge natural; i PROJECT proporciona eines de configuració dels agents pel diferents dominis d'aplicació. Actualment s'està desenvolupant la versió 2.0. Tant la documentació com el software encara no són disponibles per Internet, però aviat estaran disponibles i distribuïda per finalitats no comercials.
AgentBuilder®	http://www.AgentBuilder.com	Reticular Systems, Inc.	Java	Es un sistema integrat de desenvolupament d'aplicacions basades en agents. Es proporcionen eines visuals que fan el desenvolupament fàcil i ràpid. Es poden definir els agents, les converses que realitzen, i regles de comportament associades a un agent. Més enllà es proporcionen eines per veure el comportament de cada agent. AgentBuilder és un producte registrat però es pot obtenir una versió d'avaluació i manuals per estudiar el seu funcionament.
Architecture type-based	http://samuel.cs.uni-potsdam.de/soft/text/	University of Potsdam	Java	Entorn de desenvolupament de sistemes multiagent. Defineix un llenguatge

Development Environment (ADE)				d'interacció propi.
Bee-gent	http://www2.toshiba.co.jp/beegent/index.htm	Toshiba Corporation Systems and Software Research Laboratories	Java	Implementació d'un llenguatge interagent (ACL) basat en XML.
Bond Distributed Object System	http://bond.cs.purdue.edu	Purdue University	Java	Entorn de desenvolupament de sistemes orientats a agents. No implementen directament agents, sinó que en fan servir tècniques de comunicació.
Cybele	http://www.i-a-i.com/projects/cybele/index.html	Intelligent Automation, Inc.	Desconegut	Cybele és una infraestructura que suporta diferents serveis d'agents. Està destinat a desenvolupar agents autònoms. Inclou diferents tipus de serveis com: creació d'agents, serveis de missatges (cues, diferents tipus permesos, i diferents formes de transmissió), i eines per donar mobilitat als agents a través d'una xarxa. No hi ha cap versió disponible del software. Forma part d'un projecte del DARPA.
DECAF Agent Framework	http://www.cis.udel.edu/~graham/DECAF	University of Delaware	Java	Entorn de desenvolupament. Proporciona un entorn gràfic de desenvolupament i genera Java.
dMars	http://www.aail.oz.au/proj/dMARS-product-brief.html	Australian Artificial Intelligence Institute Ltd.	C, C++	Producte de desenvolupament i implementació orientada a agents. És un producte registrat (que per usar-lo cal comprar-lo) amb el qual s'han desenvolupat molts projectes de dominis molt diversos.
FIPA Smart	http://dirac.nosc.mil:6996/FIPA_SMART/FipaSmartIntro.html	Foundation for Intelligent Physical Agents (FIPA)	Java	Implementació de SMA fixos i mòbils basant-se en l'ACL proposat per la FIPA. La versió que d'aquest software que es pot descarregar, no permet la programació, i només inclou exemples d'utilització (és una demostració del producte).
iGENÓ	http://www.cognitive-agent.com	CHI Systems, Inc.	C/C++	iGEN TM és una eina per crear agents cognitius (tipus d'agents autònoms). Aquests agents són capaços d'identificar certs tipus de problemes, solucionar-los de la forma més eficient possible i, implementar les solucions. El sistema intenta simular el comportament i anàlisi de problemes, semblant al raonament humà.
Intelligent Agent	http://www.bitspix.com/business/main/iaf/iafintro/iafintro.html	Bits & Pixels	Java	Eina per crear agents intel·ligents fàcilment. Els agents són controlats per

Factory	iafintro/iafintro.html			regles escrites en Jess (CLIPS). Els codis dels agents són generats i requereixen ser compilats per executar-se. Es pot baixar una versió d'evaluació del sistema.
JACK Intelligent Agents	http://www.agent-software.com.au/jack.html	Agent Oriented Software Pty. Ltd.	JACK Agent Language	Entorn de desenvolupament d'agents
JAFMAS	http://www.ececs.ec.edu/~abaker/JAFMAS/	Universitat de Cincinnati	Java	JAFMAS proporciona unes llibreries de funcions escrites en Java per desenvolupar SMA. Proporciona les funcions i una metodologia de treball. Es poden implementar tècniques de negociació/cooperació. Els missatges segeixen el format de KQML.
JAM Agent	http://www.members.home.net/marcush/RS/irs.html	Intelligent Reasoning Systems	Java	Eina de desenvolupament d'agents basada en les arquitectures BDI (Belief-Desires-Intention). Es basa en les tècniques PRS (Procedural Reasoning Systems).
JATLite	http://java.stanford.edu/java_agent/html/	Stanford University	Java	Proporciona un conjunt de paquets realitzats en Java, que serveixen per desenvolupar/construir SMA. JATLite facilita la construcció d'agents que envien i reben missatges que utilitza KQML. A més proporciona altres eines per interactuar amb Internet i els seus serveis, com ara email o FTP.
Java Interagent for Multi-agent Systems (Jim)	http://www.ijia.csic.es/Projects/Jim/Jim.html	CSIC (Instituto de Inteligencia Artificial)	Java	Jim proporciona una plataforma per permetre seveis de comunicació i conversa entre un agent i una comunitat d'agents a la qual pertany. Implementa el concepte d'interagent. Cada interagent és un agent de software autònom capaç de mediar/manegar tots els missatges que es transmetin; fa de pont entre un agent de propòsit general i una comunitat d'agents.
Kafka	http://java.fujitsu.co.jp/hypertext/free/kafka	Fujitsu	Java	Kafka és una llibreria de funcions per dissenyar i desenvolupar SMA. Utilitza eines disponibles per Java com l'RMI (<i>Remote Method Invocation</i>) per transmetre objectes, i permet de desenvolupar agents fixos i mòbils. Utilitzant aquestes llibreries s'ha implementat un assistent personal (DUET).
Kasbah	http://ecommerce.media.mit.edu/Kasbah/index.html	Massachusetts Institute of Technology	Desconegut	Kasbah és actualment un projecte de recerca que permetrà de realitzar transaccions electròniques de forma segura. Es té un agent que realitza la

				mediació (segons una estratègia i una finalitat) en el sistema de comerç electrònic.
LALO	http://www.CRIM.CA/sbc/english/lalo/	Centre de Recherche Informatique de Montréal (CRIM)	LALO	LALO és un entorn de desenvolupament de SMA. LALO és un <i>agent-programming language</i> i un marc per realitzar SMA intel·ligents. Permet de crear agents reactius i deliberatius. Els fitxers font LALO són interpretats i traduïts a fitxers C++. Implementa KQML.
Multiagent Modeling LAnguage (MAML)	http://www.syslab.cer.hu/maml/	Central European University	MAML	MAML defineix unes paraules clau que defineixen l'estructura general d'una simulació. Les simulacions es realitzen en codi que s'executa sobre <i>Swarm</i> . Tot el sistema es basa en rutines escrites en Objective-C.
Multiagent Systems Tool (MAST)	http://www.gsi.dit.upm.es/~mast/			
Swarm	http://www.santafe.edu/projects/swarm/	Santa Fe Institute	C	Swarm és un software per realitzar simulacions de SMA. Està pensat especialment per a investigacions de vida artificial i implementa/estudia tota una sèrie d'interaccions entre els agents del sistema.
UMPRS Agent	http://www.members.home.net/marcush/RS/irs.html	Intelligent Reasoning Systems	C++	Eina de desenvolupament d'agents basada en les arquitectures BDI (<i>Belief-Desires-Intention</i>). Es basa en les tècniques PRS (Procedural Reasoning Systems).
Versatile Intelligent Agents (Via)	http://www.kinetoscope.com/via/default.htm	Kinetoscope Inc.	Java	Via va ser un dels primers eines completes de desenvolupament d'agents intel·ligents. Via conté tots els blocs necessaris per crear agents i donar-los tasques a realitzar. Proporciona una interfície pel desenvolupament. Es pot aconseguir versions d'avaluació per sistemes UNIX i Windows.
Zeus	http://www.ssiwyg://59/http://www.labs.bt.com/projects/agents/zeus/index.html	British Telecommunications Labs	Java	Zeus és un entorn de desenvolupament de SMA. Incorpora unes interfícies gràfiques per ajudar a l'usuari a utilitzar i dissenyar aplicacions. El sistema implementa diferents eines de col·laboració i negociació, que es basen en KQML.

6. REFERÈNCIES

- [1] Galan, Alan, "*Jafmas complete thesis*", Department of Electrical and Computer Engineering and Computer Science, 1998.
URL: <http://www.ececs.uc.edu/~abaker/JAFMAS>
- [2] Golobardes, E., Pérez, J., Porta, J.M., "*Tutorial sobre agents*", Seminari d'Intel·ligència Artificial (UPC), 1996.
- [3] Wooldridge, M., Jennings, N., "*Intelligent Agents: Theory and Practice*", Knowledge Engineering Review Volum 10 No 2, 1995.
URL: <http://www.elec.qmw.ac.uk/dai/pubs/KER95/>.
- [4] BDI Architecture, "*Belief, desire and intention architectures*".
URL: <http://hydra.elec.qmw.ac.uk/~tim/atal95/bdi.html>
- [5] Brooks, "*Subsumption Architecture Background*".
URL: http://krusty.eecs.umich.edu/cogarch3/Brooks/Brooks_Background
- [6] Foundation for Intelligent Physical Agents, "*Specification of FIPA*", FIPA, 1999.
URL: <http://fipa.org/spec/fipa99spec.htm>
- [7] Nwana, H (BT). i Wooldridge, M. (QMW), "*Software Agent Technologies*", BT Labs.
URL: http://www.labs.bt.com/agents/publish/papers/sat_report.htm
- [8] Genesereth, M. "*Knowledge Interchange Format: Specification*", American National Standard, 1995
URL: <http://logic.stanford.edu/kif/kif.html>
- [9] Finin, Tim, "*UMBC Kqml Web*", 1999.
URL: <http://www.cs.umbc.edu/agents/kqml>
- [10] FIPA, "*Relationship of ACL to KQML*", 1999.
URL: <http://dirac.nosc.mil:6996/FIPA/FIPAvsKQML.html>
- [11] Finin, T., Weber, J. [et al.], "*Specification of KQML Communication Language*", DARPA Knowledge Sharing Effort, (External Interfaces Working Group), 1993.
URL: <http://www.cs.umbc.edu>
- [12] Finin, T., McKay D., [et al.], "*KQML as an Agent Communication Language*", Conference on Information and Knowledge Management (CIKM'94), ACM Press, November 1994.
- [13] Kraus, Sarit, "*Negotiation and cooperation in multi-agent environments*", Artificial Intelligence, 94 (pags. 79-97), Ed. Elsevier, 1997.
- [14] Wooldridge, M., Jennings, N., "*Agent Techonology: Foundations, Applications, and Markets*", Ed. Springer, 1997.

7. ESTUDI ESPECÍFIC D'ALGUNES EINES

La finalitat del projecte no només és explicar per a què poden servir les implementacions de sistemes multiagent, sinó que també cal veure com es pot anar de la teoria a la pràctica.

S'ha fet un estudi en detall d'algunes de les eines que es poden trobar al mercat (Internet): com s'implementen els agents, com es comuniquen, per quins dominis d'aplicació són més indònies, etc., i a la vegada s'ha fet una comparativa entre totes.

A banda d'aquest petit conjunt, hi ha algunes eines que també són interessants i, tot i no estudiar-se en profunditat, se'n fa una descripció. Són els entorns no escollits.

7.1. Entorns escollits

Els entorns de desenvolupament que s'han triat són:

- ***JACK Intelligent Agents***, de la companyia australiana *Agent Oriented Software, Pty.Ltd.*
- ***Java-Agent Template Lite (JATLite)***, de la Univesitat d'Stanford.
- ***Java-Agent Framework MultiAgent System (JAFMAS)***, d'Alan Galan de la Universitat de Cincinnati.
- ***Zeus***, entorn de desenvolupament de sistemes multiagent creada per British Telecom.
- ***KAFKA***, de la companyia japonesa Fujitsu, proporciona una llibreria d'eines Java.

7.2. Entorns no escollits

Els entorns de desenvolupament que no s'han triat són:

- ***AgentBuilder Pro***, de la companyia americana *Reticular Systems Inc.*
- ***DECAF***, de la Universitat de Delaware.
- ***FIPA_SMART***, de la *Foundation for Intelligent Physical Agents*.

7.2.1. AgentBuilder

AGENTBUILDER

7.2.1.1. Descripció general

AgentBuilder és un entorn integrat de desenvolupament d'aplicacions de sistemes multiagent. En aquest sentit és una eina de treball molt semblant a Zeus (v. Capítol 11) , la qual estudiarem després. Permet a l'usuari dissenyar/crear qualsevol aplicació sense coneixements previs de cap llenguatge de programació. Ofereix unes opcions, a través de diferents interfícies, per configurar diferents paràmetres del domini en concret.

El desenvolupament d'una aplicació s'inicia en la interfície *Project Manager* (v. Figura 7.1). Des d'aquí podem crear, configurar, executar i monitoritzar les nostres aplicacions. Per començar a familiaritzar-nos amb la notació que es fa servir i com s'estructuren els agents i les seves funcions, podem investigar a través dels exemples que ens proporciona la versió d'avaluació descarregada, només així aprendrem a utilitzar aquest software i llurs possibilitats.

AgentBuilder estructura la metodologia de desenvolupament d'una aplicació multiagent en diferents nivells d'abstracció:

- En el nivell més alt es tenen els *projectes*. Són carpetes que estructuren les diferents aplicacions que nosaltres haguem desenvolupat.
- Dins de cada carpeta de projecte, es poden incloure una o més *agencys*. Aquest element té la missió de constituir la base en la qual crearem i s'executaran els agents. Després veurem quines característiques/funcions té.
- En el nivell més baix tenim els *agents*. Aquests elements són creats individualment i a cadascun cal assignar-lis un tipus de comportament.

Per assignar un comportament a un agent es tenen diferents eines: crear regles de comportament de la forma *precondició* → *acció*, crear estratègies de coordinació entre els agents per a dur a terme alguna acció específica (protocols), o podem crear classes d'objectes que ens interessin (i assignar-lis mètodes). Més endavant veurem com s'implementen tots aquests elements.

Aquests elements són molt interessants però la seva utilització és complicada. Com que el sistema ofereix tantes funcions, el seu aprenentatge és difícil i requereix grans dosis de paciència per utilitzar-lo.

Cal notar que aquest no és un producte de lliure distribució i cal pagar per fer-ne ús. Es pot aconseguir una versió d'avaluació acompanyada d'un manual de referència i una guia d'usuari.

7.2.1.2. L'entorn de desenvolupament

Com s'ha dit, *AgentBuilder* proporciona un entorn de desenvolupament visual i integrat, és a dir, permet de dissenyar i executar una aplicació.

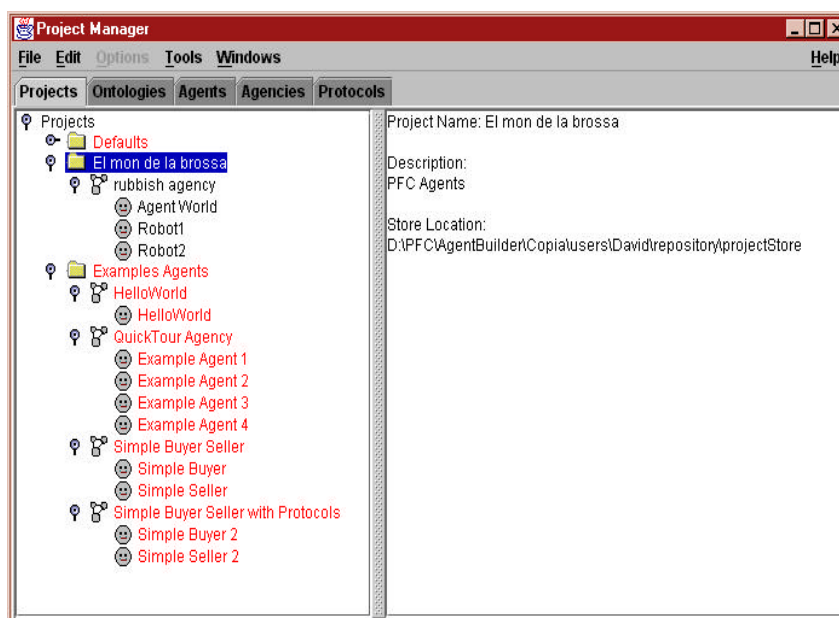


Figura 7.1: Interfície gràfica del *Project Manager* d'*AgentBuilder*. Conté tots els elements del sistema dividits en panells, i en menús desplegable a la part superior. Les aplicacions en vermell són exemples que només es poden executar i consultar, les aplicacions en negre, són creades per l'usuari.

En aquesta versió d'avaluació només podrem veure l'execució dels exemples proporcionats pel sistema (en vermell), la resta no es poden executar.

Veiem què són els diferents panells de la interfície del *Project Manager*:

- **Projects:** Permet de veure la jerarquia d'agents i agències de cada aplicació. Permet iniciar l'execució dels agents que volem.
- **Ontologies:** L'ontologia és una especificació de conceptes i defineix el tipus de coneixement del domini que necessitarà l'agent per a realitzar les seves funcions.
- **Agents:** Permet definir diferents aspectes dels agents, com ara regles de comportament que han d'incloure, ontologia a la qual està adscrit, la inclusió o no de aplicacions externes implementades a priori, definició de les accions que pot dur a terme, etc.
- **Agències:** És l'estructura que permet a diferents agents que pertanyen a una mateixa aplicació, intercanviar missatges i o realitzar accions plegats.
- **Protocols:** Definició d'estratègies que l'agent pot dur a terme. Poden importar-se estratègies dissenyades, o crear-ne de noves segons les necessitats.

7.2.1.3. Regles de comportament

Les regles de comportament són accions que l'agent porta a terme quan es donen certes circumstàncies. Cada regla té unes precondicions que han de complir-se en l'aplicació per tal de dur a terme unes accions determinades.

Per veure millor quin aspecte/forma tenen aquestes regles de comportament, es van dissenyar tres regles per simular el problema del món de la brossa, on uns robots han de recollir/anar a una posició determinada quan un agent independent (*world*) els ho mana.

Abans de detallar la implementació de les regles, cal fer notar que la transmissió de les posicions de les brosses als robots i la seva interpretació, es fa usant una estructura de dades que proporcionava el sistema. L'estructura és *PriceRequest* i conté diferents camps de diferents tipus de dades, com *strings*, *integer* i *float*.

Les regles creades són tres (v. Figura 7.2):

Crear nova brossa: L'execució dels agents pot configurar-se de manera que sigui cíclica amb un cert període. La variable *currentTime* indica que cada cop que s'iniciï l'execució de l'agent, es creï una brossa. Les coordenades de la nova brossa són aleatòries i s'inclouen en un nou registre de tipus *PriceRequest*.

Enviam la posició de la brossa: Una vegada es detecta la inserció d'un nou registre de tipus *PriceRequest* i no és cap element que s'ha enviat el mateix agent, l'enviem al destinatari. En aquest cas no es dona cap específic i es fa un *broadcast*.

Brossa rebuda: Quan es rep un missatge indicant una posició de brossa, actualitzem la posició del robot. Les variables *posx* i *posy* són internes dels robots.

7.2.1.4. Protocols o estratègies de coordinació

Les estratègies de coordinació són un element important dins els entorns de desenvolupament de SMA. En *AgentBuilder* es permet d'importar protocols ja implementats o realitzar-ne de nous.

Es té un *protocol editor* que permet de definir cada node segons sigui inicial, final o intermedi, i les diferents transicions es poden implementar en una taula de transicions.

Hi ha diferents protocols establerts com per exemple un protocol senzill de compra/venda entre dos agents, que poden ser monitoritzats visualment utilitzant una altra eina, l'*agency viewer* (v. Figura 7.3).

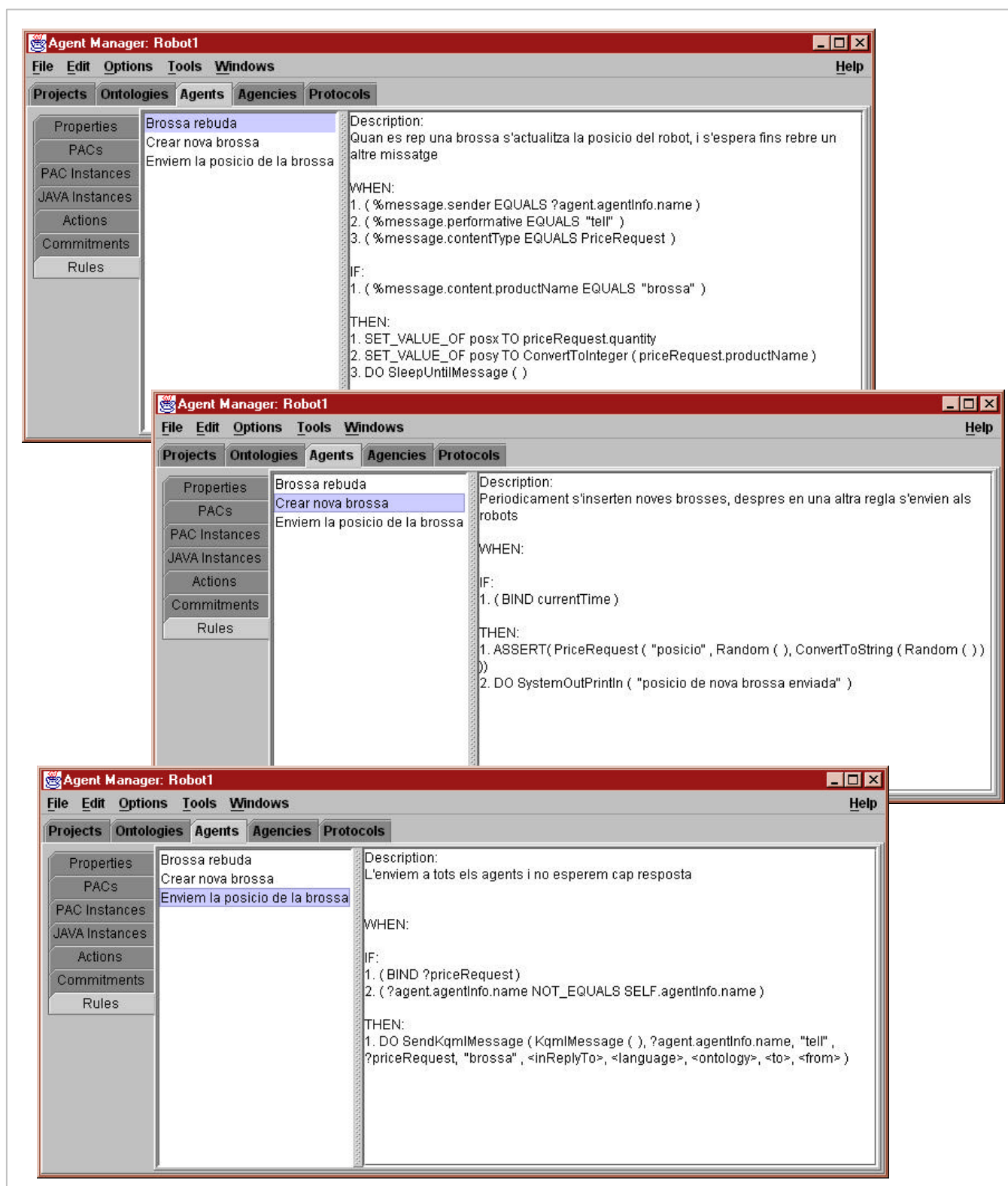


Figura 7.2: Regles creades per a l'aplicació del món de la brossa. Per crear les regles es té un editor força bo.

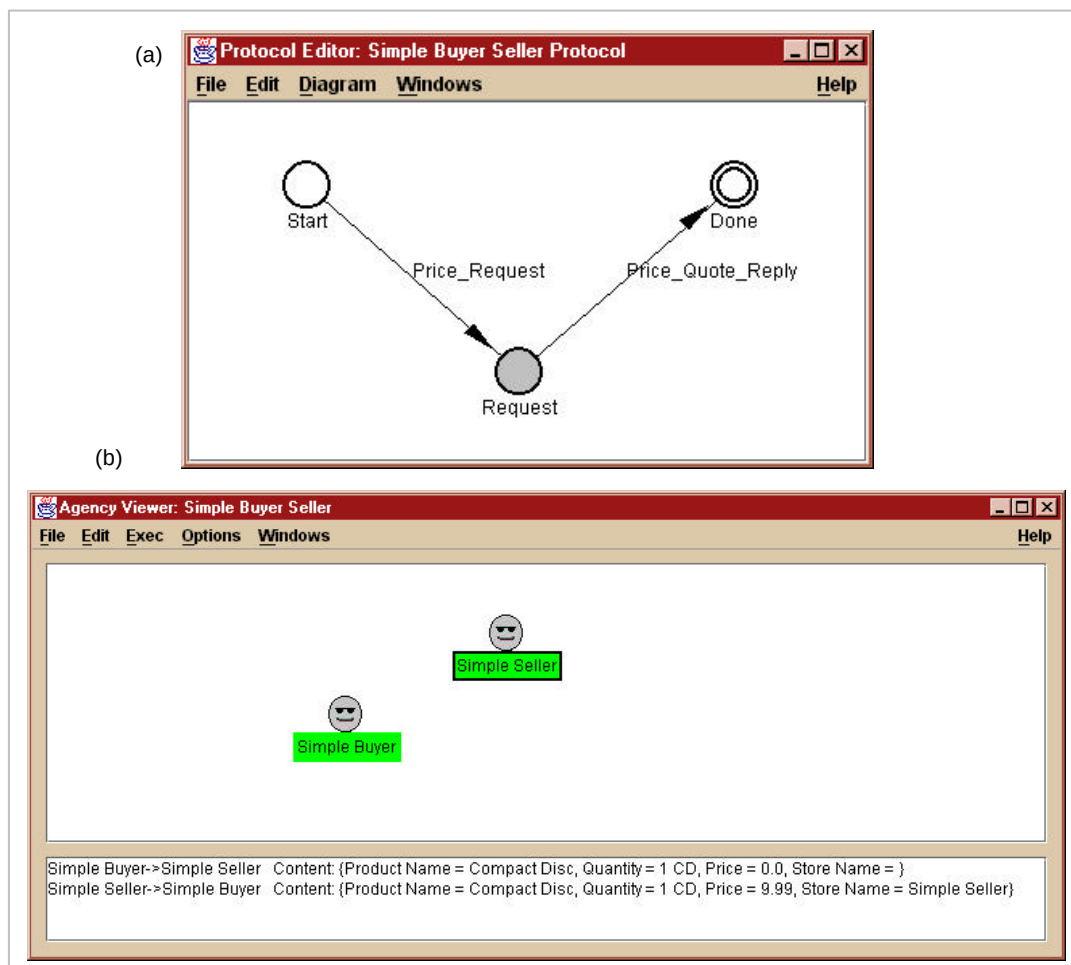


Figura 7.3: (a) Implementació d'un protocol o estratègia de coordinació.
(b) Es pot veure una simulació visual del funcionament d'aquesta estratègia.

7.2.1.5. PACs (Project Accessory Classes)

Els PACs són classes creades per a satisfer diferents funcions. Els paquets que incorpora el sistema (v. Figura 7.4) proporcionen informació de l'agent, permeten de transmetre missatges usant KQML, permeten la comunicació entre els agents i interfícies gràfiques implementades per interactuar amb l'usuari, permeten la transmissió d'objectes utilitzant RMI, i tenir referències del temps per a utilitzar en les simulacions/execucions d'aplicacions.

Altres paquets poden ser creats per l'usuari i satisfer les seves necessitats, per exemple es pot crear un paquet per accedir directament a les brosses i interactuar amb l'usuari per monitoritzar l'execució. El sistema deixa les portes obertes per a possibles ampliacions. En la versió d'avaluació no es poden implementar.

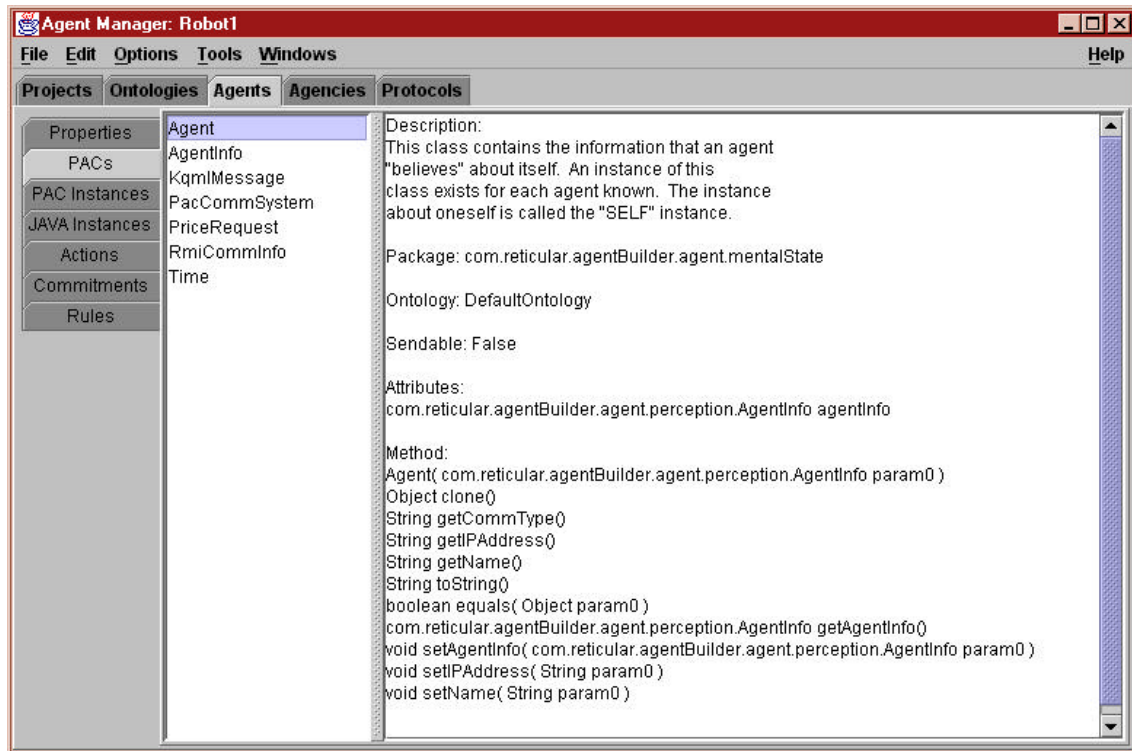


Figura 7.4: PACs que incorpora l'agent actual. Tots els agents han d'incorporar-los per poder realitzar les operacions més bàsiques. Més enllà, es poden incorporar altres, com el *PriceRequest*.

7.2.1.6. Execució

Una vegada configurats els agents, es pot passar a l'execució. L'execució dels agents es pot configurar de moltes maneres, per donar més o menys informació a l'usuari (informació interna de l'estat actual), configurar el període de simulació, per guardar informació de l'execució en fitxers auxiliars, etc. Té un disseny molt bo i pensat per a molts tipus de situacions.

Després de configurar l'execució, s'executa l'agent en una nova finestra (v. Figura 7.4). Els missatges que es mostren dependran de com haguem configurat l'execució, i es poden guardar les diferents simulacions en diferents fitxers sense cap problema.

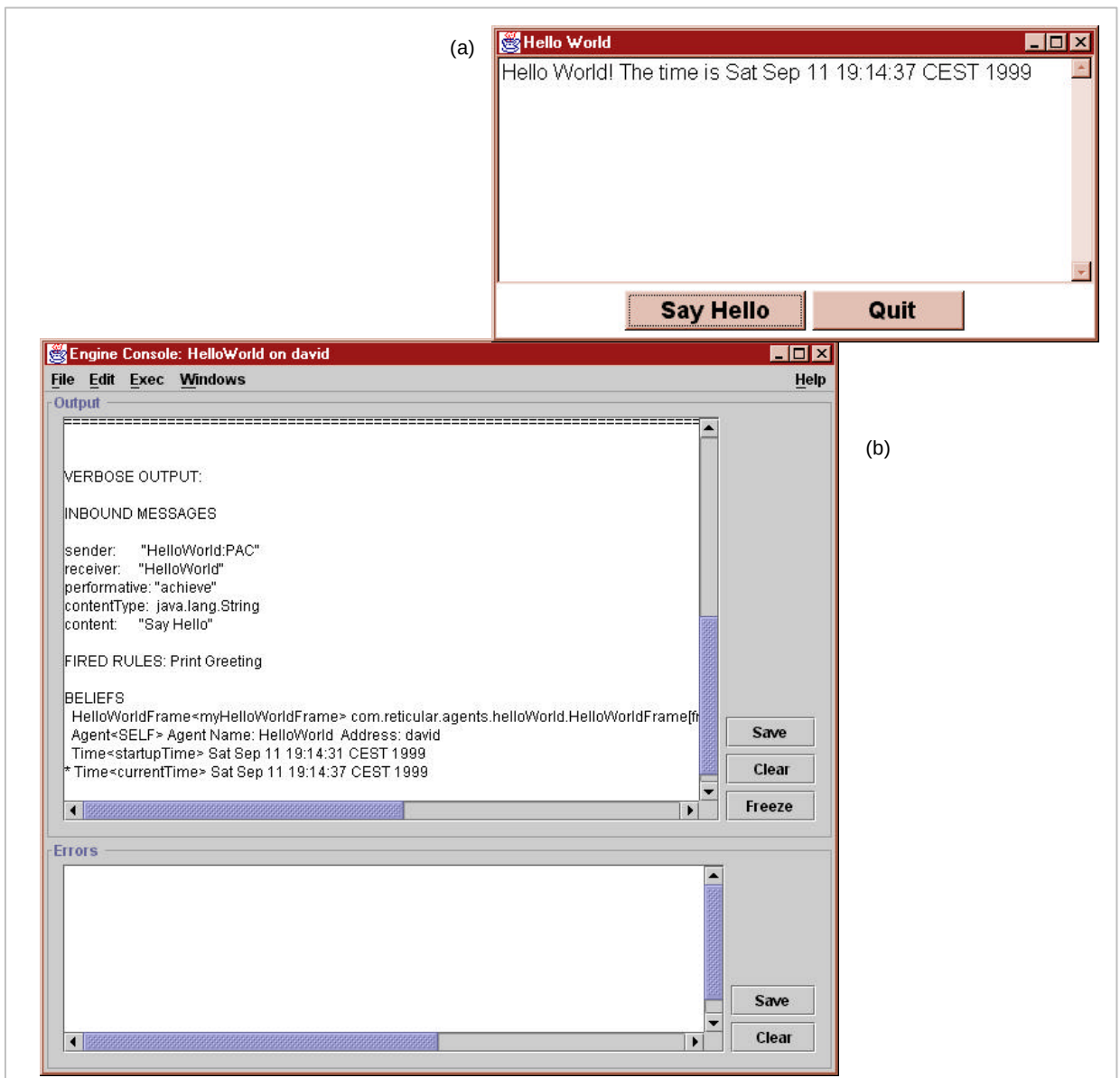


Figura 7.5: Interfície gràfica on es mostren les evolucions de l'execució.
(a) És un interfície que permet a l'usuari interactuar amb el sistema. Quan diu hola, l'agent mostra el missatge rebut.
(b) Veiem com s'activa la regla *print greeting* i ensenya el missatge rebut

7.2.2. DECAF

Aquesta és l'eina més complicada de fer anar que s'ha estudiat. La implementació i les característiques del software no semblen complicats, el què passa és que no es dona cap informació de la seva utilització, funcionament, prestacions, etc., i per avaluar s'han de fer proves a una aplicació de demostració que ens proporcionen.

El mètode de treball de Decaf és implementar aplicacions orientades a agents. És a dir es tenen aplicacions les quals fan servir tècniques per comunicar-se i cooperar com les de SMA.

La filosofia de funcionament és semblant a Zeus (v. Capítol 11). Es té una interfície gràfica que permet de crear l'entorn de l'aplicació (v. Figura 7.6). El disseny consisteix en disposar diferents blocs (que representen accions), mostrant els paràmetres d'entrada i sortida, i dibuixar gràficament les relacions que s'estableixen entre cadascun d'ells.

L'entorn es guarda en un fitxer tipus Lisp (*.lsp) i ja està.

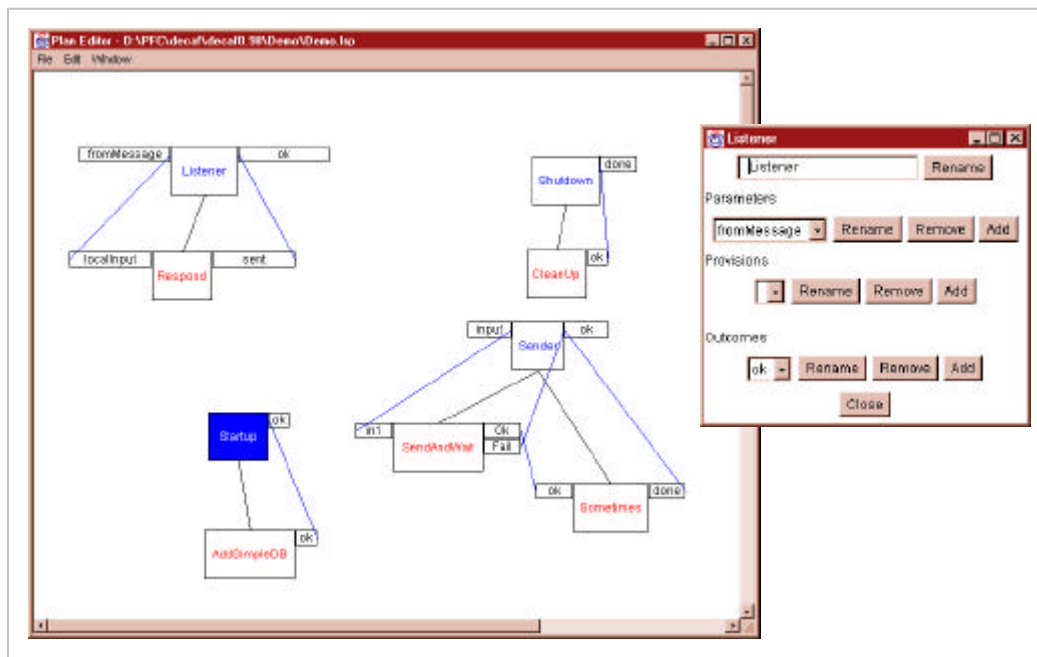


Figura 7.6: Interfície *PlanEditor* i detall de la definició d'un dels blocs (*listener*). Només donem els noms dels paràmetres.

Per executar l'aplicació es té la interfície *Agent* que després de llegir el fitxer *lsp* corresponent a l'aplicació, mostra la interfície amb la qual veurem l'execució.

L'execució la fa l'usuari enviant missatges que l'agent/aplicació interpreta i executa.

El funcionament exacte de l'aplicació *no se sap per manca d'informació dels dissenyadors*, però es veu que pot ser interessant.

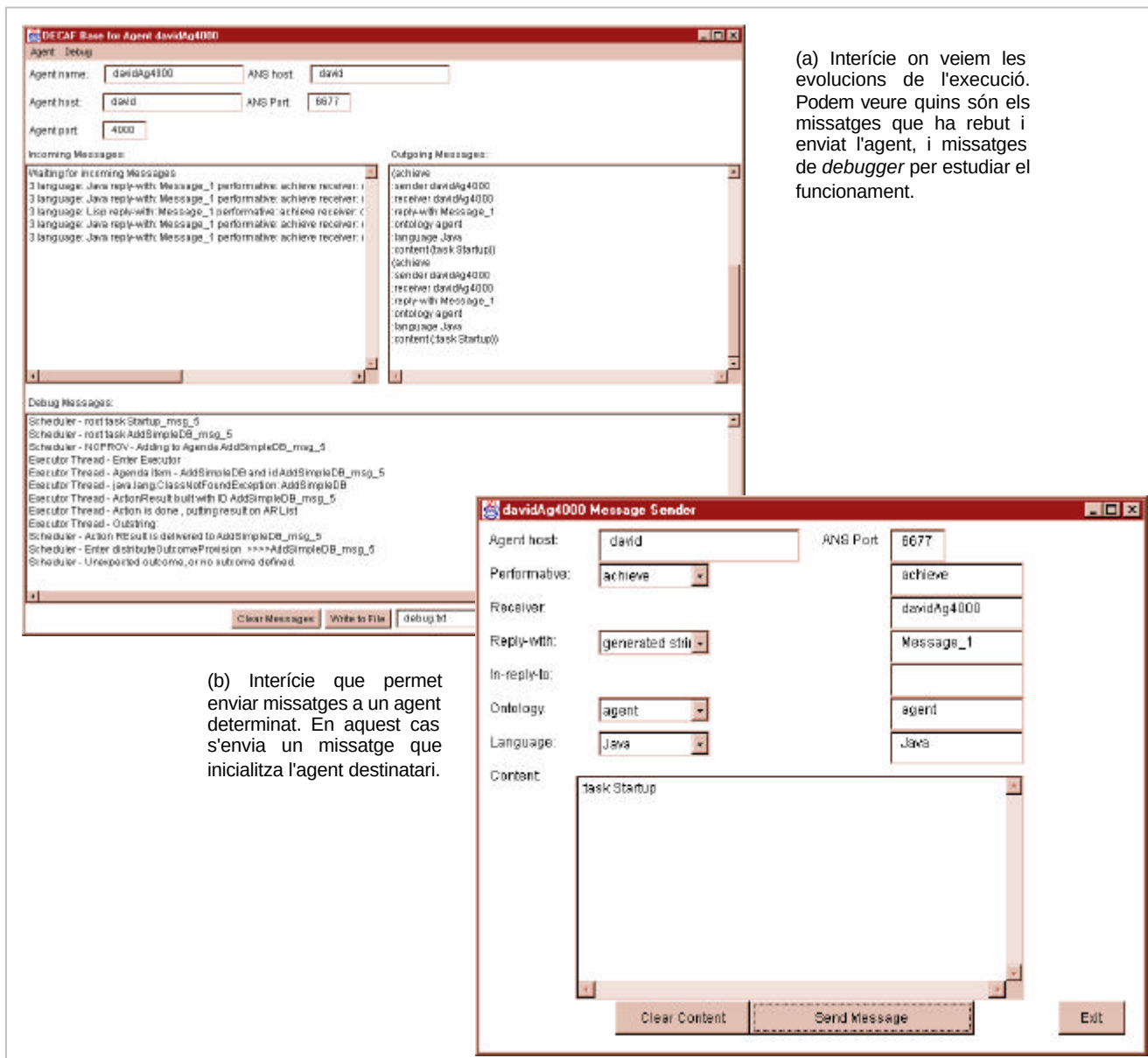


Figura 7.7: Execució d'un agent. Cal donar les ordres per a que funcioni.

7.2.3. FIPA_SMART

Totes les eines que hem estudiat que implementen algun tipus d'ACL conegut, implementen KQML. La utilització d'aquesta eina és pels usuaris que vulguin conèixer algun altre tipus d'ACL. Aquesta eina l'ha creat la *Foundation for Intelligent Physical Agents* (FIPA) i mostra algunes aplicacions de multiagent on el format dels missatges és el què ells proposen (FIPA ACL).

FIPA_SMART és una aplicació de demostració. Inclou molta documentació que explica com funciona el sistema i els diferents agents, i un glossari amb totes les accions que té el sistema donant les seves característiques, però sense donar cap tipus de detall de la seva implementació ni els codis font.

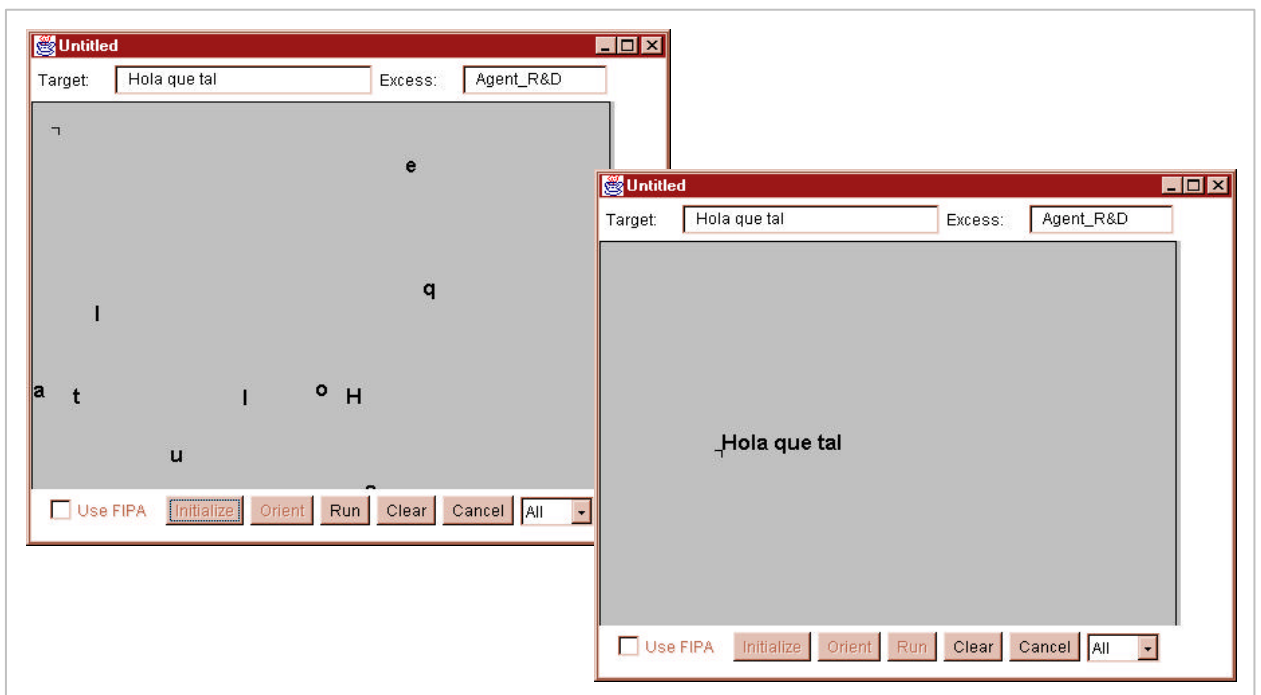


Figura 7.8: Un dels exemples que es poden provar en FIPA_SMART. Inicialment el tauler conté les lletres de la frase introduïda per l'usuari desordenades. Després s'ordenen i s'alineen a partir d'una posició indicada per l'usuari. En aquesta aplicació cada lletra és un agent i coopera amb la resta d'agents (lletres) per situar-se en ordre.

La FIPA intenta demostrar les possibilitats que té el seu producte. Per això les demostracions són força bones, i en les quals poden treballar més de 200 agents alhora (això sí, amb 128 MB de RAM i 400 Mhz de CPU).

8. ESTUDI DE JACK INTELLIGENT AGENTS

JACK Intelligent Agents

Agent Oriented Software Pty. Ltd.
<http://www.agent-software.com.au>



David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

8. ESTUDI DE JACK INTELLIGENT AGENTS.....	53
8.1. INTRODUCCIÓ	55
8.2. JACK: QUÈ ÉS UN AGENT?	55
8.3. ARQUITECTURA BDI-JACK.....	57
8.4. COMUNICACIÓ ENTRE AGENTS EN JACK	58
8.4.1. Comunicació local.....	58
8.4.2. Comunicació remota.....	58
8.5. ELEMENTS DE JACK	59
8.6. LLENGUATGE JACK (JAL)	61
8.7. TIPUS DE CLASSES JACK, INTERFÍCIES I ALTRES ELEMENTS.....	62
8.7.1. Clases principals.....	62
8.7.1.1. Agent	62
8.7.1.2. Base de coneixement	63
8.7.1.3. Event.....	65
8.7.1.4. Pla.....	66
8.7.2. Interfícies	67
8.7.3. Altres elements.....	67
8.8. APLICACIÓ D'EXEMPLE DE JACK	68
8.8.1. Codi de l'aplicació.....	69
8.8.2. Resultats.....	77
8.8.3. Càlcul del factorial d'un nombre.....	78
8.8.3.1. Codi	78
8.8.3.2. Resultats	81
8.9. APLICACIONS DE JACK	82
8.10. CONCLUSIONS	82

8.1. Introducció

JACK *Intelligent Agents* ha estat creat per Agent Oriented Software Pty. Ltd. (<http://www.agent-software.com.au>). Podem obtenir una versió d'avaluació per un temps limitat d'un més; en cas d'interessar aquest paquet cal *comprar-lo*.

JACK és un entorn de desenvolupament orientat a agents, és a dir, que proporciona les eines per tal que nosaltres creem les nostres aplicacions basant-nos en el model d'agent.

Per a ells, en concret, un agent és un sistema de software que és autònom, capaç d'actuar proactivament mentre reacciona a events en un entorn de temps real.

Bàsicament el model d'agent implementat és el *Belief Desire and Intentions* (BDI) (v. Arquitectures BDI).

La programació orientada a agents tal i com l'entèn JACK és una abstracció d'un nivell lògic (amb eines orientades a bases de coneixement i intercanvi de coneixement) i un nivell físic (proporcionant un mitjà pel pas de missatges i una diferenciació dels diferents elements que formen part del sistema) que junts formen un entorn fàcil d'entendre pels novells d'aquest tipus de programació.

Encara que aquest producte no pot ser avaluat satisfactòriament per raons que més endavant exposarem, és un producte molt interessant per projectes orientats al món multiagent, però de dimensions reduïdes, doncs té moltes eines útils per programar ràpidament però difícils de depurar en cas d'errors.

8.2. JACK: què és un agent?

Per JACK els agents són procediments (programes) autònoms que tenen uns objectius o tasques a aconseguir.

Aquests agents són preprogramats amb plans que descriuen *com* aconseguir els objectius sota les més variades circumstàncies. Així, es portaran a terme les accions tenint en compte l'estat actual del món on es treballa.

Els elements que té un agent JACK són:

- Una *base de coneixement* sobre el món (creences): poden ser dades internes que té l'agent inicialitzades arran la seva creació o dades que pot obtenir dinàmicament per intercanvi amb altres agents.
- Un *conjunt d'events* als quals pot accedir si cal: són procediment o eines de les quals pot fer-ne ús si s'escau per intentar aconseguir una fita.
- Un *conjunt d'objectius* que voldrien ser aconseguits,
- I un *conjunt de plans* que descriuen com s'ha de comportar un agent quan es produeix un event determinat.

El funcionament del sistema és simple. Un agent quan és inicialitzat, espera fins que no li donem ordres de fer alguna cosa o "veu" que ha de reaccionar a un event que s'ha produït i al qual té ordres de respondre amb alguna acció. Quan es produeix alguna d'aquestes situacions, l'agent ha de determinar quina acció prendre; busca pels plans

que puguin ser satisfactoris (alternatives) i n'aplica una que no doni problemes tenint en compte l'entorn. En cas contrari, si no es produeix cap event que li afecti, no fa res.

Com també diu el manual, a primera vista això sembla un senzill Sistema Expert (v.STRIPS,CLIPS,etc.) amb les limitacions que això implica, però es diu que la diferencia està en el comportament racional del sistema, així s'executen els plans tal com ho faria una persona racional (és l'aportació del model BDI proposat). No es trien entre les diferents opcions cegament sinó que es tenen unes propietats que donen aquest comportament humà:

- Es focalitza un objectiu, i els mètodes intenten d'aconseguir-lo.
- Un entorn de temps real on a cada moment s'escull avaluant les diferents circumstàncies que es donen.
- Un entorn de temps real de validació, on s'executen els plans mentre les condicions per ser executats són certes.
- Permet aplicacions distribuïdes.
- Poden operar cooperativament, com veurem més endavant.

Com veurem, quan hom observa la implementació d'un agent, un agent és instanciat com un objecte (perquè està implementat com una classe abstracta, i no com un *thread* o fil d'execució), llavors, com no és cap fil, només pot actuar activament (pot provocar events) quan és instanciat.

Aquest fet és important per entendre l'arquitectura BDI-JACK, doncs el motor que fa funcionar tot el sistema és la reactivitat a uns sensors, i no el propi agent amb unes funcions destinades per fer-ho.

8.3. Arquitectura BDI-JACK

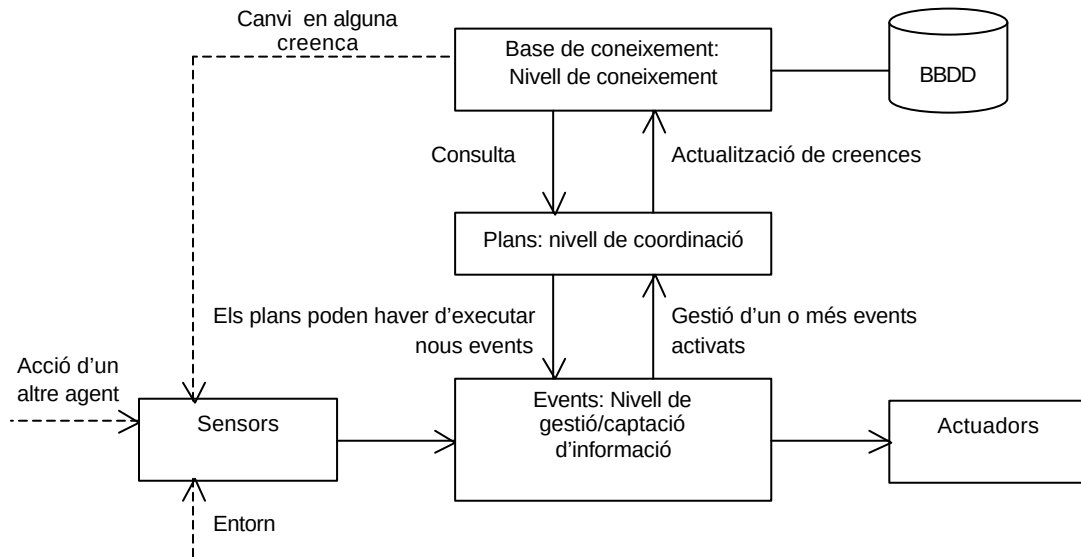


Figura 8.1: Arquitectura de Jack

L'agent en aquesta arquitectura no surt reflexat com a tal perquè JACK l'interpreta com l'*entorn* on es desenvolupa/evoluciona el sistema. És a dir, l'agent defineix les estructures de dades, els events que són possibles, i els plans que es poden fer servir, i ja està.

Conceptualment les interaccions que es donen entre els diferents nivells són clares, però a més JACK proporciona interaccions que no són directes:

- Quan en la base de coneixement hi ha un canvi en una creença, per un canvi en el món real, es pot activar un o més d'un event.
- Els events també poden ser provocats per altres *agents*, perquè cooperen, negocien, o qualsevol altre causa.
- Els events també poden ser provocats per l'entorn (qualsevol dada mesurada).

Com es veu en l'apartat següent, cadascun d'aquests elements de l'arquitectura es tracta de manera separada en JACK. Així es tindran diferents tipus de fitxers segons es tracti d'events, plans, agents o base de coneixement. Així, la dificultat d'aquest sistema està en entendre bé aquesta arquitectura i adequar-ne les nostres aplicacions.

8.4. Comunicació entre Agents en JACK

El llenguatge de comunicació entre els agents (ACL) no és estàndard, és per tant un llenguatge *ad hoc*. No hi ha implícitament implementat cap ACL, i l'ACL és implícit en el pas de missatges entre events i plans. (v. Events).

JACK proporciona facilitats per desenvolupar programació distribuïda (principalment en un entorn de xarxa). Els agents poden dirigir missatges a un altre, especificant el nom de la destinació de l'agent i, si és aplicable, el seu procés i host. L'entorn de xarxa JACK, es cuida de dirigir aquest missatge al destí desitjat.

8.4.1. Comunicació local

És el mètode que es fa servir en l'exemple anterior.

Quan es produeix una comunicació, només es necessita conèixer el nom de l'altre, doncs el receptor, quan rep un missatge, ja inclou informació sobre l'emissor.

L'agent font necessita poder enviar missatges. Per això s'inclou la declaració `#sends` i això l'habilita per poder-ho fer.

L'agent destí necessita poder manegar aquest missatge. Els plans de l'agent, envien/reben missatges que l'agent declara com a “manejables” amb la declaració `#handles`.

Quan els agents són a la mateixa màquina no és necessari el routing per la xarxa i la comunicació és local.

8.4.2. Comunicació remota.

Els elements per comunicar-se són els especificats en el punt anterior. Ara, però, entra en joc el nivell de xarxa de JACK.

El protocol de comunicació escollit és UDP/IP (protocol no orientat a connexió *User Datagram Protocol/Internet Protocol*). Per comunicar-se remotament caldrà especificar els ports origen i destí de les màquines.

8.5. Elements de JACK

Quan es va importar d'Internet aquest producte, ja se sabia que la versió del producte que s'importava era d'avaluació i no estava completa. S'han trobat a faltar alguns elements d'edició i els fitxers font d'algun element, però la resta està prou bé. No té interfície gràfica de cap mena (inicialitzar agents, definir, o monitoritzar el comportament), és a dir tot el sistema s'ha de programar escrivint-ho (la qual cosa implica una fase d'aprenentatge).

L'entorn proporciona tres elements: *JACK Agent Language* (JAL), *JACK Agent Compiler* (JAC) i *JACK Agent Kernel* (JAK).

- ➔ El primer, és el model de pseudojava que ja s'ha comentat anteriorment. Defineix una sintaxi pel desenvolupament/aprenentatge, la qual requereix del “*JAL Language Reference*” que també es proporciona amb la versió d'avaluació.
- ➔ El segon, JAC, és el compilador de JACK que dona una bona descripció dels errors que es van produint i la seva possible solució.
- ➔ El tercer element és intern de l'entorn i no es dona cap informació de la seva implementació. El Kernel proporciona nombrosos mètodes i classes internes, però no es dona en cap moment descripció d'allò que fan i com s'han d'usar. A més es dona el codi compilat (*.class) i no se'n pot analitzar el contingut. Aquest fet és important, doncs en el desenvolupament d'una nova aplicació per analitzar el funcionament de l'entorn, es van produir errors d'excepcions de les quals no es va poder concretar l'origen.

Veiem com treballa JACK fent un *make* d'una aplicació d'exemple.

Com s'ha vist un sistema d'agents té quatre tipus d'elements: plans, events, agents i base de coneixement. JACK defineix aquests elements en fitxers diferents i la manera de diferenciar-los és amb les extensions: *.plan, *.event, *.agent i *.db respectivament.

```
MS-DOS
D:\aos\aos\jack\examples\RubbishWorld>mkit

D:\aos\aos\jack\examples\RubbishWorld>java aos.jack.util.make.Main -a
The JACK make utility version 1.1
(C) Copyright 1998, Agent Oriented Software Pty. Ltd.
java aos.jack.jac.compile.Main RobotInterface.java WorldInterface.java Main.java
Rubbish.java RobotMoveEvent.event CarQueryEvent.event NewRubbishEvent.event Sta
rtSimulationEvent.event RubbishRemovalNotificationEvent.event MakeRubbishEvent.e
vent SimulationEvent.event Robot.agent World.agent MovementResponse.plan CarQuer
yResponseHandler.plan NewRubbishAppeared.plan RobotMoveRequest.plan CarQueryEval
uation.plan RubbishRemoval.plan RunSimulation.plan MakeRubbishPlan.plan
[Loaded plugin map file aos\jack\jac\plugin\map]
Writing to file D:\aos\aos\jack\examples\RubbishWorld\RobotMoveEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\CarQueryEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\NewRubbishEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\StartSimulationEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\RubbishRemovalNotification
Event.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\MakeRubbishEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\SimulationEvent.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\Robot.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\World.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\MovementResponse.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\CarQueryResponseHandler.ja
va
Writing to file D:\aos\aos\jack\examples\RubbishWorld\NewRubbishAppeared.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\RobotMoveRequest.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\CarQueryEvaluation.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\RubbishRemoval.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\RunSimulation.java
Writing to file D:\aos\aos\jack\examples\RubbishWorld\MakeRubbishPlan.java
*** Compilation Finished ***
*** in 16920 ms ***
javac RobotInterface.java WorldInterface.java Main.java Rubbish.java RobotMoveEv
ent.java CarQueryEvent.java NewRubbishEvent.java StartSimulationEvent.java Rubbi
shRemovalNotificationEvent.java MakeRubbishEvent.java SimulationEvent.java Robot
.java World.java MovementResponse.java CarQueryResponseHandler.java NewRubbishAp
peared.java RobotMoveRequest.java CarQueryEvaluation.java RubbishRemoval.java Ru
nSimulation.java MakeRubbishPlan.java

D:\aos\aos\jack\examples\RubbishWorld>
D:\aos\aos\jack\examples\RubbishWorld>
```

La primera part de la compilació dóna els errors de sintaxi i en cas d'haver-hi s'atura aquesta. Si no hi ha errors, com en el cas de la figura, es transcriuen els fitxers **.agent*, **.plan*,... en fitxers **.java*. Aquesta fase pot durar alguns minuts.

Passada aquesta fase tenim fitxers Java que es compilen amb *javac*. Aquesta fase pot donar errors, doncs la interpretació que fa JACK a voltes no pot ser interpretada pel compilador de Java, llavors cal refer el fitxer original de manera que s'eviti l'error.

Si aquestes fases tenen èxit només cal executar el fitxer que inicia el sistema.

8.6. Llenguatge JACK (JAL)

JAL defineix un conjunt d'extensions de Java. Els termes en negreta són paraules clau de JAL, o elements imprescindibles per la utilització del JACK. Veiem-les en un exemple:

```
import java.util.Vector;
import aos.jack.jak.core.*;
import Car;
import Rubbish;

agent Robot extends Agent implements RobotInterface {

    #handles event NewRubbishEvent;
    #handles event RobotMoveEvent;
    #handles event CarQueryEvent;
    #handles event SimulationEvent;

    #uses plan NewRubbishAppeared;
    #uses plan MovementResponse;
    #uses plan CarQueryResponseHandler;
    #uses plan RunSimulation;

    private Vector RubbishAvailable;
    Rubbish bestRubbish;
    int myLane;
    int myDisplacement;

    public Robot (String n) {
        super (n);
        RubbishAvailable = new Vector(10,5);
        bestRubbish=null;
        myLane=0;
        myDisplacement=0;
        System.out.println("*** Robot "+n+ " activat i Funcionant.");
    }

    public static void main (String [] args) {
        Jak.initialize(args);
    }

    public void setBestRubbish( Rubbish r ) {
        System.out.println("El robot vol agafar...");
        bestRubbish = r;
        bestRubbish.printInfoRubbish();
        System.out.println("aquesta.");
    }
    public boolean moreRubbish() {
        if (bestRubbish==null) return false;
        return true;
    }

    public Rubbish getBestRubbish() {
        return (Rubbish)bestRubbish;
    }
    public int getLane() {
        return myLane;
    }
    public int getDisplacement() {
        return myDisplacement;
    }
    public void die() {
        System.exit(1);
    }
    public void updatePosition(int l,int d) {
        myLane=l;
        myDisplacement=d;
    }
}
```

8.7. Tipus de classes JACK, interfícies i altres elements

8.7.1. Classes principals

JAL introdueix quatre tipus de classes principals:

- Agent.
- Base de coneixement (BC).
- Event.
- Pla.

8.7.1.1. Agent

Com s'ha dit, l'agent és una classe abstracta que és instanciada com nosaltres vulguem. Defineix l'entorn el domini de l'agent amb les estructures de dades que té i amb les funcions permeses.

Les estructures de dades, poden ser estructures simples com vectors, matrius, elements unitaris (enters, caràcters, constants, ...), o una base de coneixement (BC) associada a un agent. Com més endavant veurem, aquesta BC és un vector de tuples amb una relació d'ordre i unes operacions associades.

Les funcions que pot realitzar són el tractament d'events (així es defineix quin/s event/s pot manejar l'agent), els plans possibles, i les accions que pugui definir l'agent per a determinades accions.

L'estructura general d'un agent és:

```
agent nom_agent extends Agent {  
  
    //plans, events, BC que manega l'agent  
    #private database FactorialDb known(); //defineix la BC FactorialDb amb l'alias known()  
    #handles event FactorialGoal; //l'agent pot manegar l'event FactorialGoal  
    #uses plan FactorialPlan; //l'agent pot manegar el pla FactorialPlan  
  
    //possibles accions que podem prendre  
    #posts event FactorialRequest compute; //en aquest cas, diem que dins l'agent es pot  
    enviar/rebre un event del tipus FactorialRequest (l'alias és compute)  
    #sends event ServerPlan request; //habilitem a l'agent per tal d'enviar un missatge del  
    tipus ServerPlan.  
  
    //després de les inicialitzacions de les possibilitats de l'agent, podem definir les  
    estructures de dades que necessitem:  
    Vector vector;  
    public int i,j;  
    static int ID=33;  
  
    //Codi de les funcions que es vulgui implementar:  
    public FactorialAgent(String s)  
    {  
        super(s);  
    }  
}
```

```

public long factorial(int i)
{
    FactorialRequest r = compute.factorial(i);
    if (postEventAndWait(r))
        return r.result;
    return -1;
}

```

NOTA: la implementació no és exacta i està feta per il·lustrar com es defineixen els agents. Aquesta implementació està basada en l'exemple del càlcul d'un factorial que es proporciona com a exemple de demostració.

Les accions que es defineixin a l'agent, per poder ser accedides des de plans cal que estiguin definides en un fitxer d'interfície (v. Interfícies (apartat 8.7.2)).

8.7.1.2. Base de coneixement

La base de coneixement (BC) és potser la part que fa més interessant aquest entorn de desenvolupament.

Aquesta BC ha estat especialment concebuda per la implementació d'agents, i dóna unes funcions molt interessants:

- Hi ha un manteniment automàtic de la consistència lògica i les restriccions, segons la clau.
- Hi ha un manteniment de les creences, segons es tracti d'una semàntica en un món tancat (*closed-world*)⁷ o obert (*open-world*)⁸
- Es poden programar events quan s'efectua alguna operació sobre la BC.

Les operacions permeses en aquestes BC són *afegir (assert)* i *treure (retract)* –només en un món tancat -. Les accions que nosaltres vulguem realitzar sobre la BC les haurem d'escriure.

Totes les actualitzacions, consultes o esborrats de creences (registres) de la BC (BBDD), realitzen un recorregut registre a registre per aquesta. JACK proporciona eines per tal d'optimitzar els recursos i dóna dues possibilitats a l'hora de crear les relacions en la BC:

- Lineal, de manera que donant un patró, aquest s'anirà buscant linealment a través de la BC fins a trobar-lo.
- Indexada, de manera que internament es manté una estructura d'índexs per la/es clau/s i les cerques són més eficients.

⁷ Un món tancat (*closed-world*) segueix una lògica booleana i només permet que una creença sigui certa o falsa. Assumeix que *tot* és conegut.

⁸ Un món obert (*open-world*) modela el coneixement d'un món real. Permet tres tipus d'estats en una creença: cert (*true*), fals (*false*) o no se sap (*unknown*).

El programador és qui, preveient la càrrega de la relació de la BC, tria quin mètode pot anar millor, doncs per una càrrega baixa no val la pena tenir índexs, mentre que si la càrrega pot ser alta, el recorregut lineal és ineficient.

Definició de la BC:

```
database objectes_habitacio extends ClosedWorld {
    #key field String nom_objecte;
    #value field String tipus_objecte;
    #value field int color;
    #value field int fila;
    #value field int columna;

    #indexed query consulta_pel_nom(logical String nom_objecte, String
tipus_objecte, int color, int fila, in columna);
    #indexed query consulta_pel_tipus(String nom_objecte, logical String
tipus_objecte, int color, int fila, in columna);
}
```

a l'agent s'ha de definir l'ús d'aquesta BC:

```
#private database objectes_habitacio obj_bc;
```

i després, en els plans podem executar consultes, insercions, etc.

```
→ if(obj_bc.consulta_pel_nom(patro_nom, t,f,c)) { accio_cerca_trobada; }
→ obj_bc.assert(n0,t0,f0,c0);
→ obj_bc.retract(n0,t0,f0,c0);
```

Figura 8.2: La BC, definició i operacions que es poden realitzar.

Les operacions permeses sobre la BC permeten una gestió òptima. Se'ns permet d'usar la BC amb les operacions que es defineixen per defecte: afegir, consulta i esborrat de creences, però també podem redefinir-les per tal que es facin events/operacions que nosaltres desitgem.

8.7.1.3. Event

Un event JAL és com una funció o declaració de serveis. Els events contenen prou informació per a definir com pot ser requerit el servei i com pot funcionar.

És a dir, un event quan es crida no fa res més. La feina la fan els plans que el maneguen.

JACK fa servir els events per realitzar la comunicació dins els agents i entre agents. Llavors, implícitament tenen la funció de transmetre dades, així l'ús més habitual dels events és:

- No fer res, llavors el pla implícitament no necessita res més i sap el que ha de fer (ex. un event que sigui d'apagar la llum; quan el pla que el gestiona el rep, aquest sap que ha de fer i no necessita cap més coneixement)
- O, servir de pont entre qui genera l'event i el pla/ns que el gestionen, i alhora que comparteixin/transmetin informació. En aquest cas els events requereixen el pas de paràmetres que contenen la informació a transmetre. Aquests paràmetres són tipus de dades estàndards: enters, cadenes, vectors, matrius, i fins i tot, objectes propis de l'agent.

```
import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;
import aos.apib.*;
```

```
message Ping extends MessageEvent {
    #posted as ping() {
    }
}
```

Exemple d'event senzill, que només serveix per advertir a un pla de la seva presència i aquest (el pla) ja sap què ha de fer.

En aquest cas, hi ha intercanvi de dades.

El pla que gestioni l'event pot consultar el contingut de les variables globals a les que se'ls ha assignat el valor del paràmetre de qui els ha generat.

En l'exemple el pla receptor sap que la brossa que estava a (l,d) ja s'ha recollit.

```
import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;
```

```
event RubbishRemovalNotificationEvent extends MessageEvent {
    public int lane;
    public int displacement;

    #posted as removeRubbish(int l, int d) {
        System.out.println("RubbishRemovalNotification.event");
        System.out.println("Esborrada brossa (lina,dist): (" + l + ", " + d + ")");
        lane=l;
        displacement=d;
    }
}
```

Figura 8.3: Exemples d'events JACK.

Dins dels events hi ha un tipus força interessant: *BDIGoalEvent*. Aquest tipus d'events es tracten com objectius que quan són activats, vol dir que s'han aconseguit. Aquests tipus d'events tenen unes operacions especials que permeten la sincronització amb els plans que els gestionen.

8.7.1.4. Pla

Els plans en l'arquitectura BDI-JACK s'han situat en un nivell de coordinació. Bàsicament tenen la missió de gestionar tots els events que es puguin donar, i a la vegada interaccionar amb la BC (v. Apartat Base Coneixement).

Els plans s'executen responent a events. Hi ha el problema de resoldre conflictes quan hi ha més d'un pla que pot executar-se alhora. En aquestes situacions JACK dóna dues possibles solucions:

Abans del cos (*body*) del pla, podem posar les precondicions que volem que es compleixin per executar el pla:

- Si les precondicions només depenen de l'execució d'events, podem comparar el tipus d'event que volem manejar, i si es vol el valor de paràmetres. Si totes les comparacions són certes, es passarà a executar el pla.
Fem servir el mètode `static boolean relevant(EventType event)`.
- Si les precondicions depenen de l'entorn (de les creences, dels objectius, ...) JACK també dóna una solució. Les condicions que podem considerar són de diversos tipus: consultes en la BC, condicions lògiques, etc.
En aquest cas fem servir el mètode `context()`.

Dins el cos del pla, es fan les operacions que es creguin necessàries. Els plans són l'element més actiu de tota l'arquitectura JACK. Es poden fer totes les operacions que qualsevol pot pensar:

- es poden consultar valors de paràmetres dels events,
- es pot assegurar que un objectiu sigui aconseguit abans de continuar (*insist, test, achieve*),
- es poden crear nous events, ja sigui asíncronament (*post/send*) o síncronament (*subtask*).
- es poden executar mètodes que l'agent defineixi com d'ús públic,
- es poden enviar events a altres agents, i cooperar o negociar,
- fer consultes/actualitzacions sobre una BC,
- posar-se a dormir,
- mantenir un control sobre uns objectius (*achieve, insist, test*).

8.7.2. Interfícies

Un agent pot opcionalment implementar una interfície. Aquest fet fa que el codi dels agents pugui ser reutilitzable segons es vulgui. Aquesta interfície proporciona una zona comuna entre els agents que els permeten compartir plans.

Normalment quan un agent executa un pla, aquest pla cridarà altres mètodes per tal d'implementar les funcions del nivell més baix. Aquests mètodes no apareixen en la definició de pla; s'espera d'ells que estiguin definits en qualsevol lloc de l'agent.

La definició d'una interfície per part d'un agent, fa que aquest sigui "propietari" dels mètodes que s'hi defineixen i prevé de conflictes entre mètodes compartits.

8.7.3. Altres elements

Una de les principals avantatges d'utilitzar JAL és el seu vincle pròxim a Java. Això fa d'ell que sigui fàcil aprendre JAL si ja es té un coneixement previ de Java. També podem usar aquest fet en el desenvolupament d'un sistema.

Un exemple d'aquest "intercanvi", podem trobar-lo si hem d'usar un tipus d'objecte determinat, amb unes característiques pròpies. Llavors, tenim dues alternatives: usar les eines que proporciona JAL i definir una base de coneixement; o fent servir les eines de Java, definim una classe per l'objecte determinat, i uns mètodes per interactuar amb les seves característiques. Aquest fet fa que es pugui aprofitar tota la potència de la POO.

8.8. Aplicació d'exemple de JACK

En l'exemple es tenen fragments d'un cas pràctic implementat: *El Món de la Brossa*. En aquest sistema tenim un entorn (món) i un robot.

El robot que té la missió de recollir la brossa que va apareixent, és un agent.

El món té la missió de controlar tot el que passa, va creant brosses.

El funcionament de Jack, situació→acció, fa que hagi gran nombre de fitxers. Per il·lustrar al lector de la seva relació, es pot observar l'esquema següent:

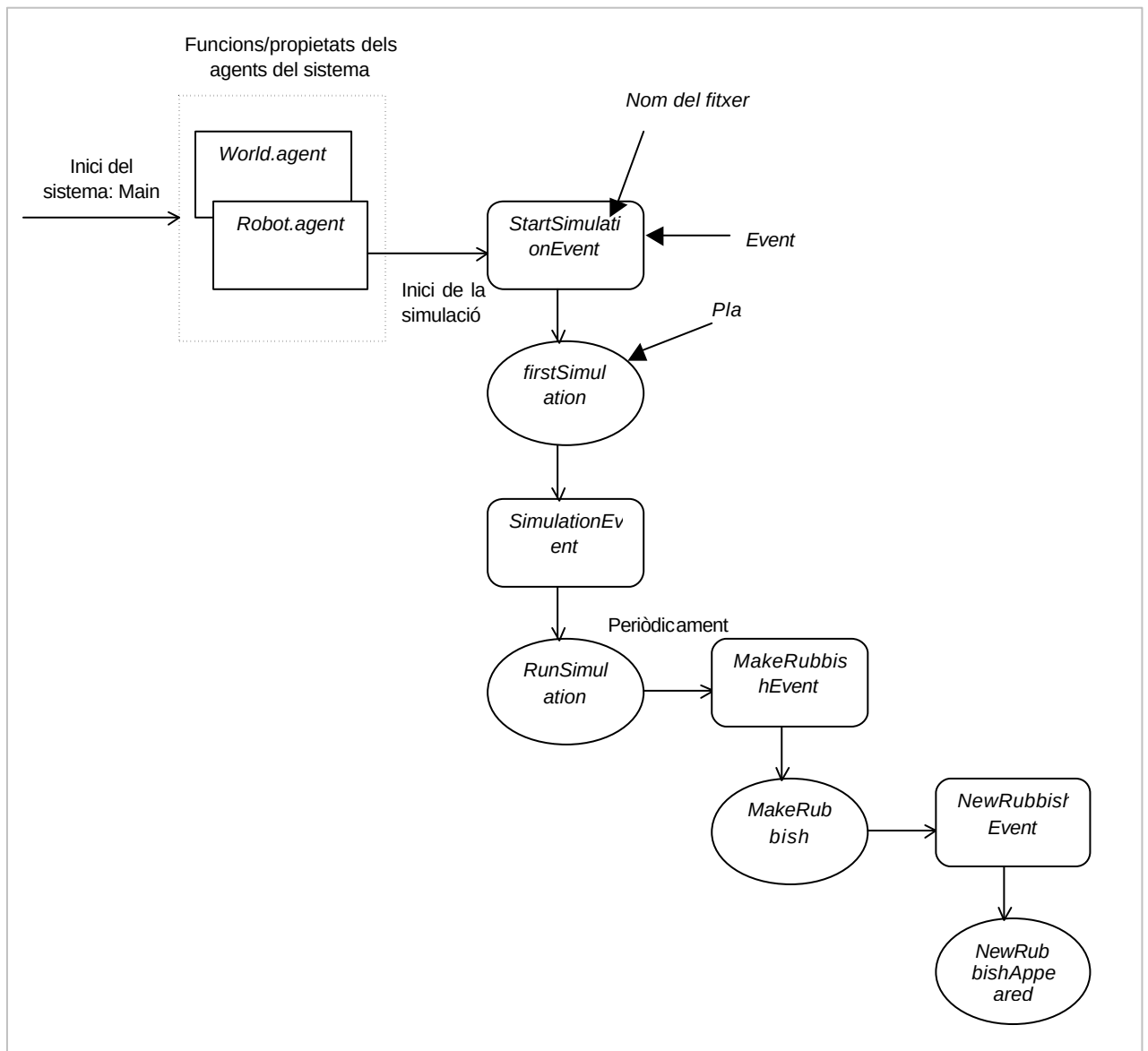


Figura 8.4: Seqüència de funcionament/activació d'events de l'aplicació.

8.8.1. Codi de l'aplicació

```
//*****
// File: /mnt/d/pfc/aos/RubbishWorld/Main.java

import aos.jack.jak.core.*;
import java.io.*;

class Main {
    public static void main(String [] args) {
        args = Jak.initialize(args);
        System.out.println("Benvinguts al MON de la BROSSA.");
        new World ("Street");
        new Robot ("R2D2");
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/Robot.agent

import java.util.Vector;
import aos.jack.jak.core.*;
import Rubbish;

agent Robot extends Agent implements RobotInterface {

    #uses plan NewRubbishAppeared;
    #uses plan MovementResponse;
    #uses plan RunSimulation;
    #uses plan firstSimulation;

    #handles event NewRubbishEvent;
    #handles event RobotMoveEvent;
    #handles event SimulationEvent;
    #handles event StartSimulationEvent;

    #posts event StartSimulationEvent ssim;

    private Vector RubbishAvailable;
    Rubbish bestRubbish;
    int myLane;
    int myDisplacement;

    public Robot (String n) {
        super (n);
        RubbishAvailable=new Vector(10,5);
        bestRubbish=null;
        myLane=0;
        myDisplacement=0;

        System.out.println("Robot activat esperant instruccions");
        postEvent(ssim.startSim());
        System.out.println("Robot funcionant");
    }

    public static void main (String [] args) {
        Jak.initialize(args);
        //new Robot("robot");
        System.out.println("Agent ROBOT creat. Pero esta dormit!!");
    }

    public void setBestRubbish( Rubbish r ) {
        System.out.println("El robot vol agafar...");
        bestRubbish = r;
        bestRubbish.printInfoRubbish();
        System.out.println("aquesta.");
    }

    public boolean moreRubbish() {
        if (bestRubbish==null)
            return false;
        return true;
    }
}
```

```

    }

    public Rubbish getBestRubbish() {
        return (Rubbish)bestRubbish;
    }

    public int getLane() {
        return myLane;
    }

    public int getDisplacement() {
        return myDisplacement;
    }

    public void die() {
        System.exit(1);
    }

    public void updatePosition(int l,int d) {
        myLane=l;
        myDisplacement=d;
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/World.agent

import java.util.Vector;
import java.util.Random;
import aos.jack.jak.core.*;
import Rubbish;

agent World extends Agent implements WorldInterface{

//    #uses plan RobotMoveRequest;
//    #uses plan RunSimulation;
//    #uses plan firstSimulation;

//    #handles event RobotMoveEvent;
//    #handles event NewRubbishEvent;

//    #handles event SimulationEvent;
//    #handles event StartSimulationEvent;

//    #sends event NewRubbishEvent newRubbish;
//    #sends event StartSimulationEvent startSimulation;

    Vector rubbish;
    int iteracio=0;

    public World (String n1) {
        super(n1);
        System.out.println("Hola. "+n1+" està viu. Tot correcte.");
        rubbish = new Vector (10,5);
        //postEvent(startSimulation.startSim());
        System.out.println("World ha despertat Robot?? 0 no!!");
    }

    public static void main (String [] args) {
        Jak.initialize(args);
        System.out.println("WORLD creat i inicialitzat");
    }

    public Rubbish makeRubbish() {
        Random rand = new Random ( (long)System.currentTimeMillis() );
        int rubLane = Math.abs(rand.nextInt() % 10);
        int rubDisp = Math.abs(rand.nextInt() % 100);

        Rubbish newRub = new Rubbish(rubLane, rubDisp);
        rubbish.addElement(newRub);
        //postEvent(newRubbish.newRubbish(rubLane,rubDisp)); //Passem
(fila,despl)
        return newRub;
    }
}

```

```

    public void removalRubbish(int l, int d) {
        Rubbish victim = new Rubbish(l,d);

        rubbish.removeElement(victim);

        System.out.println("La brossa ...");
        victim.printInfoRubbish();
        System.out.println("ha estat esborrada, ara
tenim" + (String)rubbish.toString());
    }

    public Rubbish getClosest(int l, int d) {
        int closest=0, distance;
        Rubbish rubTemp;

        System.out.println("Busquem la brossa més propera...");
        rubTemp = (Rubbish)rubbish.elementAt(0);
        int closest_distance=(rubTemp.getLane()-l)+(rubTemp.getDisplacement()-d);
        for ( int i = 0 ; i < rubbish.size() ; i++) {
            rubTemp = (Rubbish)rubbish.elementAt(i);
            distance=(rubTemp.getLane()-l)+(rubTemp.getDisplacement()-d);
            if (distance < closest_distance) {
                closest=i;
            }
        }
        rubTemp.printInfoRubbish();
        System.out.println("és la brossa més propera");
        return(Rubbish)rubbish.elementAt(closest);
    }

    public int getClosest(int l) {
        int closest=0, closest_distance=100, distance;
        Rubbish rubTemp;

        for (int i=0 ; i<rubbish.size() ;i++) {
            rubTemp=(Rubbish)rubbish.elementAt(i);
            if(rubTemp.getLane()==l) {
                distance=rubTemp.getDisplacement();
                if (distance<closest_distance) {
                    closest=i;
                }
            }
        }
        rubTemp=(Rubbish)rubbish.elementAt(closest);
        return rubTemp.getDisplacement();
    }

    public void getInfoRubbish() {
        Rubbish rubTemp;
        if(rubbish.isEmpty())
        { System.out.println("No hi ha elements"); }
        else
        {
            System.out.println("Recorrem el vector de brosses actives");
            for ( int i = 0 ; i < rubbish.size() ; i++) {
                rubTemp = (Rubbish)rubbish.elementAt(i);
                rubTemp.printInfoRubbish();
            }
            System.out.println("Acabat el recorregut.");
        }
    }

    public void setIteracio(int n) {
        iteracio=n;
    }

    public int getIteracio() {
        return iteracio;
    }
}

```

```

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/firstSimulation.plan

import aos.jack.jak.core.*;
import java.util.Random;

plan firstSimulation extends Plan {

    #handles event StartSimulationEvent ev;
    #sends event SimulationEvent sim;

    static boolean relevant (StartSimulationEvent ev)
    { return true; }

    body() {
        System.out.println("Som dins de firstSimulation.plan");
        @post(sim.Sim());
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/StartSimulationEvent.event

import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;

event StartSimulationEvent extends MessageEvent {

    #posted as startSim() {
        System.out.println("StartSimulation.event ha rebut notificacio.");
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/SimulationEvent.event

import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;

event SimulationEvent extends Event {

    #posted as Sim() {
        System.out.println("SimulationEvent.event ha estat activat.");
    }
}

```

```

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/RunSimulation.plan

import aos.jack.jak.core.*;
import java.util.Random;

plan RunSimulation extends Plan {

    #handles event SimulationEvent ev;

    #uses agent implementing WorldInterface world;
    #uses agent implementing RobotInterface robot;

    #sends event NewRubbishEvent newRub;

    //static boolean relevant (SimulationEvent ev)
    //{ return true; }

    body() {
        System.out.println("Som dins de RunSimulation.plan");
        int ID_estat = world.getIteracio();
        System.out.println("Estat simulacio (estat)="+ID_estat);

        if (ID_estat==0) {
            System.out.println("Comencem la simulacio");
            world.setIteracio(ID_estat+1);
            @post(ev.Sim());
        }
        else {
            System.out.println("Actualitzant el sistema ...");

            int l = robot.getLane();
            int d = robot.getDisplacement();
            System.out.println("Posicio robot(fila,desp):("+ l +","+ d +")" );

            if (robot.moreRubbish()) {
                Rubbish rub=world.getClosest(l,d);
                int y1=rub.getDisplacement();
                int x1=rub.getLane();
                int x=l;
                int y=d;
                System.out.println("Anem a recollir la brossa de
"+x1+","+y1+"\ni som a "+x+","+y);
                int dx=x1-x;
                int dy=y1-y;
                int absdx=Math.abs(dx);
                int absdy=Math.abs(dy);

                if ((dx==0) && (dy==0))
                {
                    world.removalRubbish(x1,y1);
                }
                else if (dx==0)
                    robot.updatePosition(x,y+(dy/absdy));
                else if (dy==0)
                    robot.updatePosition(x+(dx/absdx),y);
                else if (absdx<absdy)
                    robot.updatePosition(x+(dx/absdx),y);
                else if (absdx>=absdy)
                    robot.updatePosition(x,y+(dy/absdy));

                System.out.println("MOGUTS A "+robot.getDisplacement()+
", "+robot.getLane());
            }
            else {
                System.out.println("No hi ha brossa.");
            }

            Random rand = new Random ( (long)System.currentTimeMillis() );
            int n = Math.abs(rand.nextInt() % 10);
            System.out.println("El num. que ha sortit es: "+n);
            if (n>9) {
                System.out.println("RunSimulation ha creat una brossa...");
                @post(newRub.newRubbish());
                //world.makeRubbish();
            }
            world.getInfoRubbish(); //veiem on son les brosses...
        }
    }
}

```

```

        world.setIteracio(ID_estat+1);
        @post(ev.Sim());
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/MakeRubbishEvent.event

import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;

event MakeRubbishEvent extends MessageEvent {

    #posted as make() {
        System.out.println("MakeRubbish posted");
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/MakeRubbishPlan.plan

import aos.jack.jak.core.*;

plan MakeRubbishPlan extends Plan {

    #handles event MakeRubbishEvent newRub;
    #uses agent implementing WorldInterface world;

    body() {
        System.out.println("Ordre de crear una nova brossa.");
        world.makeRubbish();
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/NewRubbishEvent.event

import aos.jack.jak.core.*;
import aos.jack.jak.event.*;
import java.io.*;

event NewRubbishEvent extends MessageEvent {

    #posted as newRubbish() {
        System.out.println(" newRubbishEvent posted");
    }
}

```

```

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/NewRubbishAppeared.plan

import aos.jack.jak.core.*;
import Rubbish;

plan NewRubbishAppeared extends Plan {

    #handles event NewRubbishEvent newRubbish;
    #uses agent implementing RobotInterface robot;
    #uses agent implementing WorldInterface world;

    body() {

        System.out.println("Ordre de crear una nova brossa.");
        Rubbish newRub = world.makeRubbish();
        System.out.println("New Rubbish Arrived. Robot Ready For Action");
        Rubbish r = robot.getBestRubbish();

        if (r==null) {
            System.out.println("No existing Rubbish - I like this one");
            robot.setBestRubbish(newRub);
        }
        else {
            int dist1=newRub.distanceFrom(robot.getLane(),robot.getDisplacement());
            int dist2=r.distanceFrom(robot.getLane(),robot.getDisplacement());

            if (dist1<dist2) {
                System.out.println("New Rubbish nicer than old");
                robot.setBestRubbish(newRub);
            }
            else
                System.out.println("New rubbish not as nice as old");
        }
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/RobotInterface.java

import java.util.Vector;
import Rubbish;

interface RobotInterface {
    public void setBestRubbish(Rubbish r);
    public int getLane();
    public int getDisplacement();
    public Rubbish getBestRubbish();
    public boolean moreRubbish();
    public void updatePosition(int l, int d);
    public void die();
}

```

```

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/Rubbish.java

//Definició de la classe Rubbish;
import java.lang.Math;

public class Rubbish {
    int lane;
    int displacement;

    Rubbish(int l, int d) {
        lane=l;
        displacement=d;
    }

    //Mètodes de CONSULTA:
    int getLane() {
        return lane;
    }
    int getDisplacement() {
        return displacement;
    }

    //Mètodes d'ASSIGNACIÓ:
    void setLane(int l) {
        lane=l;
    }
    void setDisplacement(int d) {
        displacement=d;
    }

    void setLaneDisplacement(int l, int d) {
        lane=l;
        displacement=d;
    }

    //Mostrar INFORMACIÓ:
    void printInfoRubbish() {
        System.out.println("Info Rubbish (lane,disp)="+lane+", "+displacement+"");
    }

    //Càlcul de distàncies entre (lane,displacement) i (x,y):
    int distanceFrom(int x, int y) {
        double a = Math.abs(y-lane);
        double b = Math.abs(x-displacement);
        return (int)Math.sqrt(a*a+b*b);
    }
}

//*****
// File: /mnt/d/pfc/aos/RubbishWorld/WorldInterface.java

import aos.jack.jak.plan.*;
import aos.jack.jak.event.*;
import aos.jack.jak.agent.*;
import aos.jack.jak.core.*;
import Rubbish;

public interface WorldInterface {
    public Rubbish makeRubbish();
    public void removalRubbish(int l, int d);
    public Rubbish getClosest(int l, int d);
    public int getClosest(int l);
    public void getInfoRubbish();

    public void setIteracio(int n);
    public int getIteracio();
}

```

8.8.2. Resultats

El codi del paràgraf anterior és l'enèsima versió que s'ha realitzat d'aquesta aplicació. Aquesta versió, igual que les precedents, no han aconseguit funcionar completament. El llenguatge JAL fa una traducció a fitxers Java. L'error detectat, sempre es localitza al mateix lloc i es dona en el fitxer traduït. Aquest error es dona tant en un sistema Linux com en un sistema Windows, i no se sap com solucionar-lo.

Els resultats que s'obtenen d'executar el codi del paràgraf anterior són:

```
d:\aos\RubbishWorld\java Main
Benvinguts al MON de la BROSSA.
Agent Street@%portal:
    Event NewRubbishEvent does not have any relevant plans
Hola. Street està viu. Tot correcte.
World ha despertat Robot?? 0 no!!
Robot activat esperant instruccions
StartSimulation.event ha rebut notificacio.
Robot funcionant
Som dins de firstSimulation.plan
SimulationEvent.event ha estat activat.
Task terminated because of uncaught exception
    Exception was java.lang.ClassCastException: Robot
    Initiated by SimulationEvent:80d13df:0:FILTERED
    thrown by SimulationEvent:80d13df:0:FILTERED
        NO STATE

java.lang.ClassCastException: Robot
    at RunSimulation.<init>(RunSimulation.java:43)
    at RunSimulation.<init>(RunSimulation.java:110)
    at RunSimulation.createPlan(RunSimulation.java:129)
    at aos.jack.jak.event.Event.c01$0Bb(XXXX.java:1)
    at aos.jack.jak.event.Event.run(XXXX.java:1)
    at aos.jack.jak.task.Task.rC1BCXM(XXXX.java:1)
    at aos.jack.jak.agent.Agent.OrWR$Qo(XXXX.java:1)
    at aos.jack.jak.agent.Agent.xMqR$Qx(XXXX.java:1)
    at aos.jack.jak.agent.Agent.run(XXXX.java:1)
    at aos.jack.jak.scheduler.qRNmmPq.run(XXXX.java:1)
```

Funciona fins que s'hauria d'activar el pla *RunSimulation.plan*.

Abans d'aquest error, veiem com es van activant els diferents plans, events i agents, tal com es mostra a l'esquema de funcionament que es va donar. El procés seguit fins llavors és similar a seguir un recorregut en un graf, on els arcs són els events, i els nodes són els plans.

Aquest resultat és frustrant doncs s'ha realitzat tot un procés de desenvolupament i no s'ha obtingut cap resultat positiu. Tot i que aquest exemple no ha funcionat correctament, el sistema té unes possibilitats i per veure-ho mostrarem un exemple proporcionat per l'entorn JACK que funciona.

8.8.3. Càlcul del factorial d'un nombre

Veiem els resultats d'una aplicació desenvolupada pels dissenyadors de JACK. L'exemple és el càlcul d'un factorial.

Tot i la simplicitat del codi, s'utilitzen diferents elements del sistema que són molt interessants, com ara una interfície, pas de missatges i la utilització d'una Base de Coneixement.

8.8.3.1. Codi

```
//*****  
// File: D:\aos\aos\jack\examples\Test.java
```

```
public class Test {  
    public static void main(String [] arg)  
    {  
        FactorialAgent fa = new FactorialAgent("albert");  
  
        for (int i=1; i<9; i++)  
            System.out.println("Factorial of "+i+" is "+fa.factorial(i));  
  
        System.exit(0);  
    }  
}
```

Es calcula el factorial de 0..i, on i=8. Cada crida calcula el factorial i-èssim.

```
//*****  
// File: D:\aos\aos\jack\examples\FactorialAgent.agent
```

```
agent FactorialAgent extends Agent {  
  
    #private database FactorialDb known();  
  
    #posts event FactorialRequest compute;  
  
    public  
    FactorialAgent(String s)  
    {  
        super(s);  
    }  
  
    public long  
    factorial(int i)  
    {  
        FactorialRequest r = compute.factorial(i);  
  
        if (postEventAndWait(r))  
            return r.result;  
        return -1;  
    }  
  
    #handles event FactorialGoal;  
    #handles event FactorialRequest;  
    #uses plan ServerPlan;  
    #uses plan FactorialPlan;  
}
```

Això inicia el primer *ServerPlan*.

```

//*****
// File: D:\aos\aos\jack\examples\FactorialDb.db

database FactorialDb extends ClosedWorld {
    #key field int value;
    #value field long factorial;

    #indexed query get(int v, logical long f);
    #indexed query get(logical int v, logical long f);
}

//*****
// File: D:\aos\aos\jack\examples\FactorialGoal.event

event FactorialGoal extends BDIGoalEvent {
    int n;
    logical long result;

    #posted as
    factorial(int n, logical long r)
    {
        this.n = n;
        result = r;
    }
}

//*****
// File: D:\aos\aos\jack\examples\FactorialRequest.event

event FactorialRequest extends Event {
    int n;
    long result;

    #posted as factorial(int n)
    {
        this.n = n;
    }
}

//*****
// File: D:\aos\aos\jack\examples\FactorialPlan.plan

plan FactorialPlan extends Plan {
    #handles event FactorialGoal ev;

    #modifies database FactorialDb known;
    #posts event FactorialGoal compute;

    body()
    {
        System.out.println(" [computing "+ev.n+"!]");
        if (ev.n <= 1) {
            ev.result.unify(1);
        } else {
            logical long result;

            @achieve(known.get(ev.n-1, result),
                compute.factorial(ev.n-1, result));
            ev.result.unify(result.as_long() * ev.n);
        }
    }

    #reasoning method
    pass()
    {

```

La BC és un món tancat.
Cada registre de la BC conté l'índex del
factorial i el factorial calculat.

Si no hi fós, no podríem fer
@achieve(cond, GoalEventName).

Avalua la condició: si és *true*, l'objectiu
està aconseguit i continua amb el pla; si
és *false*, l'objectiu no està aconseguit i
cal fer una subtasca per fer-ho possible.

Si no es troba ev.n-1
(factorial anterior a
l'objectiu), cal calcular-lo:
compute.factorial(e
.n-1, result).

```

    known.assert(ev.n, ev.result.as_long());
  }
}

```

```

//*****
// File: D:\aos\aos\jack\examples\ServerPlan.plan

```

```

plan ServerPlan extends Plan {
  #handles event FactorialRequest ev;

  #reads database FactorialDb known;
  #posts event FactorialGoal compute;

  body()
  {
    logical long result;
    @achieve(known.get(ev.n, result),
             compute.factorial(ev.n, result));
    System.out.println(ev.n+"! = "+result.as_long());
    ev.result = result.as_long();
  }
}

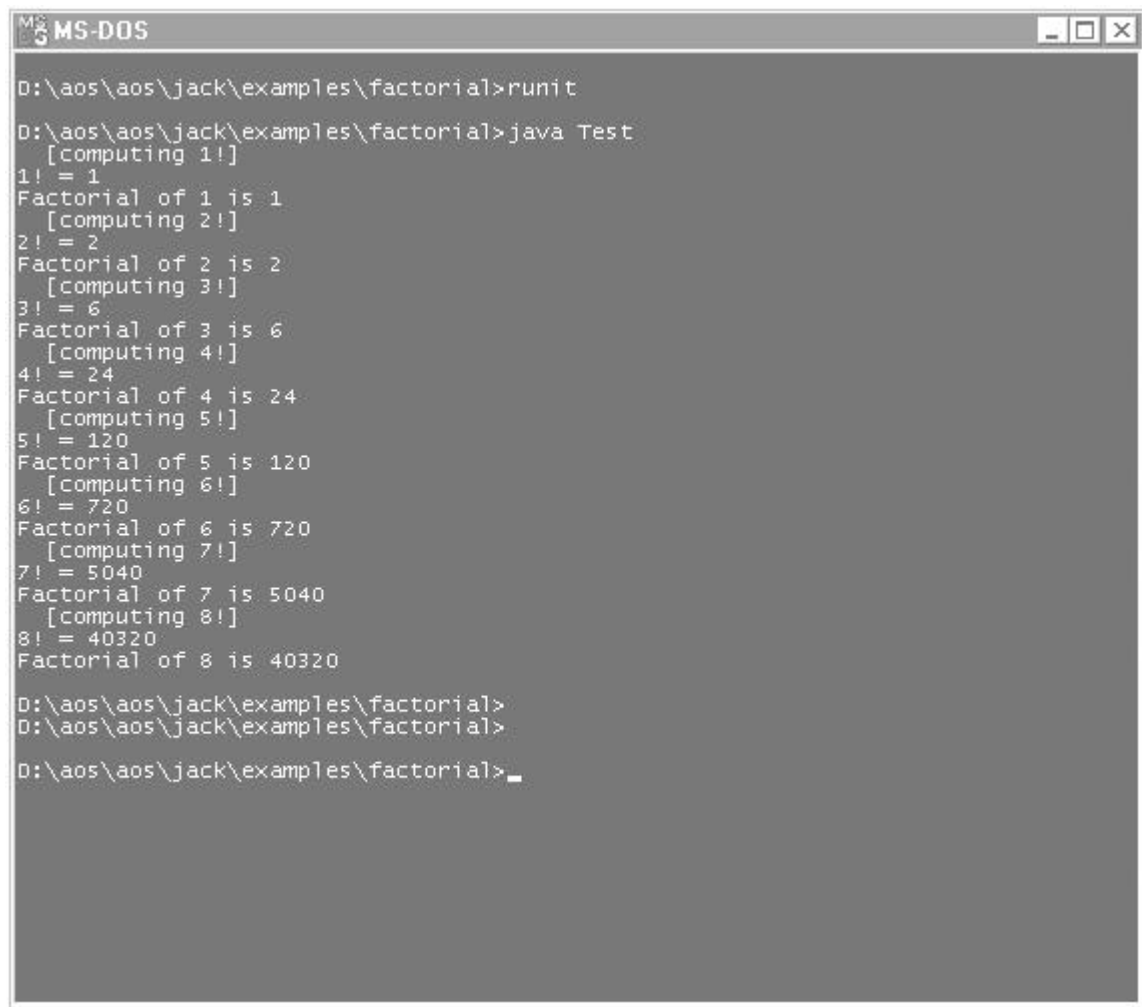
```

Inicia el càlcul del factorial. Comença amb compute.factorial(n,result), on n és l'índex del factorial desitjat i result el resultat (inicialment a 1).

Quan arriba aquí, ja ha acabat.

8.8.3.2. Resultats

Una vegada compilats els fitxers JAL, es generen uns fitxers Java que són compilats. Una vegada realitzat aquest procés i no es troben errors, s'inicia l'execució. En el nostre cas, s'ha implementat un fitxer Java (*Test.java*) que conté el cos principal i és el que inicialitza el sistema i interpreta els resultats.



```
MS-DOS
D:\aos\aos\jack\examples\factorial>runit
D:\aos\aos\jack\examples\factorial>java Test
[computing 1!]
1! = 1
Factorial of 1 is 1
[computing 2!]
2! = 2
Factorial of 2 is 2
[computing 3!]
3! = 6
Factorial of 3 is 6
[computing 4!]
4! = 24
Factorial of 4 is 24
[computing 5!]
5! = 120
Factorial of 5 is 120
[computing 6!]
6! = 720
Factorial of 6 is 720
[computing 7!]
7! = 5040
Factorial of 7 is 5040
[computing 8!]
8! = 40320
Factorial of 8 is 40320
D:\aos\aos\jack\examples\factorial>
D:\aos\aos\jack\examples\factorial>
D:\aos\aos\jack\examples\factorial>
```

Figura 8.5: Càlcul del factorial de 8. Podem veure com per calcular el factorial de 8, s'han calculat els factorials anteriors (d'1 a 7).

En la Figura 8.5, es pot veure el resultat de l'execució del càlcul del factorial de 8.

8.9. Aplicacions de JACK

Les aplicacions de JACK han de ser en dominis senzills, sense masses elements. S'ha pogut veure com en una petita aplicació com la del *món de la brossa* s'han implementat nombrosos elements.

Una de les aplicacions més directes que poden veure's és la implementació de SMA reactius. Amb el sistema JACK es poden realitzar aplicacions de control de plantes, de cadenes de muntatge d'indústries,

8.10. Conclusions

Havent vist altres eines de desenvolupament de sistemes multiagent, es poden valorar més els aspectes positius, i fer notar també els negatius, d'aquest sistema.

Un aspecte positiu és el tipus d'arquitectura i la implementació que se n'ha fet. És de valorar l'esforç en crear una arquitectura BDI, amb unes funcions força bones. Cal fer notar que no hi ha masses productes d'aquesta mena.

Un aspecte positiu són les facilitats que es donen per aprendre'l i avaluar-lo. Per *baixar-lo* d'Internet s'ha de registrar i obtenir un password. Aquest procés va ser molt ràpid i amb instruccions precises dels passos a seguir. A més, el codi del programa anava acompanyat d'uns exemples (bàsics) i d'un manual d'usuari bastant complet.

Un aspecte negatiu és la falta d'eines d'ajut al programador i a l'usuari, per tal d'usar, crear, o monitoritzar el sistema. Altres entorns proporcionen interfícies gràfiques per ajudar tant al programador com a l'usuari.

Un altre aspecte negatiu és la falta de transparència en el codi implementat. És a dir, encara que no es doni el codi font de les classes, a l'especificació de les funcions caldria donar informació dels paquets Java que s'estan fent servir, per llavors el programador poder fer la tria de les opcions que més bé li vagin, segons les circumstàncies.

Finalment cal notar que aquest producte s'ha de *comprar*; no hi ha versions de lliure distribució per científics ni estudiants, i el producte que vam baixar *legalment* no es pot modificar ni reprogramar, només es pot *veure* per un mes. Passat aquest període d'avaluació, els propietaris del producte envien una notificació per dir-nos que s'ha acabat el temps i que si volem continuar usant-lo, cal comprar-lo. El *kernel* de JACK té coneixement de la data d'alta i caducitat, i passada aquesta última, no deixa compilar res.

9. ESTUDI DE JATLITE BETA

Java-Agent Template Lite Beta (JATLite)

<http://www.agent-software.com>



David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

9.	ESTUDI DE JATLITE BETA	83
9.1.	DESCRIPCIÓ.....	85
9.2.	JATLITE: QUÈ ÉS UN AGENT?	87
9.3.	ARQUITECTURA DE JATLITE.....	89
9.3.1.1.	Abstract layer	90
9.3.1.2.	Base layer	91
9.3.1.3.	KQML Layer.....	92
9.3.1.4.	Router Layer.....	92
9.3.1.5.	Protocol Layer.....	92
9.4.	DESENVOLUPAMENT D'UNA APLICACIÓ JATLITE.....	93
9.4.1.	<i>Descripció del desenvolupament</i>	94
9.4.1.1.	Connexions: medi de transmissió de missatges	94
9.4.1.2.	Creació/funcionament dels agents.....	97
9.4.1.3.	Comunicació.....	99
9.4.1.4.	Tipus d'accions de comunicació o performatives	100
9.4.1.5.	Esquema del sistema	101
9.4.2.	<i>Codi font de l'aplicació</i>	103
9.4.2.1.	Cos principal.....	103
9.4.2.2.	Interfície gràfica de l'agent	107
9.4.2.3.	Interfície gràfica de la zona de treball	113
9.5.	POSSIBLES APLICACIONS DE JATLITE	115

9.1. Descripció

Java-Agent Template Lite (JATLite) és un conjunt de paquets de Java que facilita el desenvolupament de sistemes d'agents. JATLite proporciona eines de comunicació bàsiques (*templates*) basades en TCP/IP. Especialment basa aquesta comunicació en la implementació de l'ACL KQML.

Una manera còmoda d'usar/aprendre la sintaxi és veure'l de forma modular, començant pel nivell més bàsic on es defineixen les estructures i funcions bàsiques, i anar avançant en funcions i mètodes més generals.

Implementa tot un sistema de comunicacions: establir/mantenir una connexió entre dos agents, permet realitzar negociacions/cooperacions per resoldre alguna tasca, intercanviar fitxers via FTP, o enviar/rebre emails.

JATLite defineix una manera de treballar. Cal un temps per aprendre, i a més no hi ha cap interfície gràfica dissenyada per a ajudar en el desenvolupament. Tampoc, la informació que es proporciona (tutorials, manuals, etc.) no és abundant. Només s'adjunta una descripció de les classes (JavaDoc), quatre exemples i l'explicació que tothom pot trobar a la pàgina web d'on es descarrega el paquet.



Tot i aquests petits detalls (que caldria solucionar per part dels desenvolupadors), JATLite, quan hom aprèn a usar-lo, és una potent eina per crear agents.

JATLite és una de les eines més “famoses” dins el món del desenvolupament de SMA, i fins i tot té una llista d'usuaris on es poden intercanviar coneixements.

A més la descàrrega/instal·lació d'aquest paquet és molt senzilla: podem descarregar els fitxers d'instal·lació via FTP de l'adreça <ftp://cdr.stanford.edu/pub/JATLite>. En aquest lloc es poden trobar diferents fitxers per a diferents màquines (UNIX, Win), i no són gaire grans. L'ús segons consta als exemples és restringit. Com és un producte de caràcter científic, l'autor ha publicat la seva adreça per facilitar els tràmits d'ús/manipulació del seu software: <http://cdr.stanford.edu/html/people/jeon-bio.html>.

Un fet que és destacable és la llibertat que es deixa al programador per desenvolupar una aplicació. JATLite només dóna algunes funcions que hom pot implementar, però la resta es cosa nostra, i cal tenir coneixements del llenguatge de programació Java per fer-ho.

Les característiques principals de JATLite són:

Aplicable a	Aplicació. Applet.
Processat de missatges	Síncrona. Asíncrona.
ANS	Suportat pel Router
Comunicació	Establiment/manteniment de connexions
Comunicació amb agents foranis	Parcialment suportat. Els missatges dels agents foranis, han d'esser intel·ligibles pels agents JATLite.
Repartiment de missatges	Enquesta. Cues de missatges.
Router	Suportat. Peça imprescindible per la comunicació.
Socket	Persistent amb Timeot.
No KQML	Suportat, però KQML es defineix per defecte com ACL del sistema.
Seguretat	Proporcionada: establiment de connexió amb user i password. Opcional.
FTP online	Suportat.
SMTP online	Suportat.

JATLite proporciona una manera per tal que un agent a través d'un applet pugui connectar amb un altre agent. També es pot fer servir JATLite de manera aïllada en aplicacions concretes.

El processat dels missatges, basats en TCP/IP, pot ser síncron o asíncron. La sincronització és difícil d'aconseguir, però JATLite processa els missatges en ordre cronològic d'arribada a mida que van arribant d'un mateix agent. Es poden realitzar processats de missatges de forma asíncrona si aquests missatges provenen de diferents agents.

L'ANS *Agent Name Service* proporciona l'encaminament dels missatges cap als agents del seu sistema. Aquest servei el proporciona el *Router*. Cada agent no necessita ni recordar ni mantenir totes les adreces dels possibles receptors els quals poden anar canviant. L'ANS pot proporcionar informació si els agents li demanen.

Pel manegament de missatges es proporciona polling (enquesta) i cua. Quan un agent espera enviar un missatge a un altre agent, l'emissor mirarà si hi ha una connexió disponible. Si la connexió no està establerta, l'agent emissor intentarà connectar amb l'agent receptor i, si es té èxit, s'enviarà el missatge. Això és un *message polling* (intents repetitius d'enviar un missatge) però això pot tenir problemes quan la connexió amb el receptor no pot ser iniciada. Llavors, l'emissor necessitarà una cua de missatges per a guardar i un mecanisme per a enviar els missatges quan la connexió estarà establerta. El *Router* manté una cua de missatges i un mecanisme per a enviar els missatges automàticament tan bon punt les connexions estan establertes.

El pas de missatges es fa via sockets. Cada socket està gestionat (escoltar/enviar) per un thread o fil d'execució. El timeout es fa servir a nivell xarxa per tal d'assegurar que totes les connexions estiguin actives.

El ACL (Llenguatge de comunicació entre agents) per defecte és KQML. JATLite proporciona un nivell de la seva arquitectura dedicat a implementar-lo. Si no el volguéssim fer servir, podríem implementar-ne un altre amb les estructures/funcions de

comunicació que ens proporcionen els nivells inferiors de JATLite. No és una tasca senzilla.

Pel que fa a seguretat, JATLite proporciona funcions senzilles com la identificació de l'usuari i el password, afegint tantes funcions més sofisticades com es vulguin.

JATLite ofereix funcions FTP (*File Transfer Protocol*). Els missatges KQML es poden contenir dins de fitxers i transmetre'ls usant aquest protocol de transport. El receptor el podrà rebre fent una connexió FTP.

També podem embolcallar els missatges en correus electrònics usant el protocol SMTP (*Simple Mail Transfer Protocol*).

9.2. JATLite: què és un agent?

Malgrat la diversitat de definicions que hi ha d'agent, JATLite és especialment dissenyat per utilitzar en desenvolupament d'una infraestructura *Typed-Message Agents*, definit en termes de comunitat d'agents. La comunitat ha d'intercanviar els missatges per tal de complir una tasca (v. Figura 8.1). Per proporcionar l'entesa entre els agents, el format dels missatges i la seva semàntica poden estar establerts (usant un ACL usat en altres dominis) o no (usant un ACL, definit a propòsit per l'aplicació/sistema). JATLite utilitza un ACL àmpliament usat que és KQML (*Knowledge Query and Manipulation Language*); el format i utilització d'aquest ACL s'ha explicat anteriorment dins aquesta documentació.

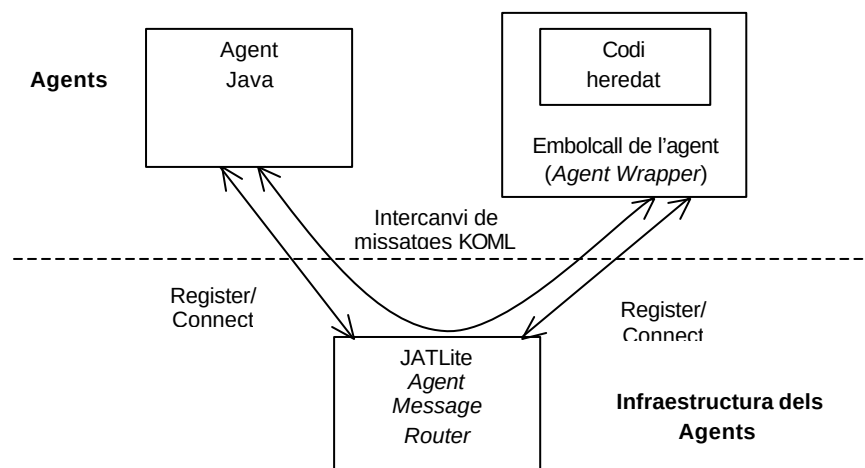


Figura 8.1: JATLite: Infraestructura d'encaminament de missatges.

La semàntica necessita que el protocol de transport no sigui només client-servidor sinó del tipus parell-parell. Un mòdul de software individual no és un agent del tot si no pot comunicar-se amb els altres candidats a agent només amb un protocol client-servidor, sense degradació de les tasques col·lectives. Així, els agents poden cooperar els uns amb els altres per a solucionar un problema i poden transmetre missatges dels sense esperar el requeriment del client; aquest esquema correspon generalment al concepte d'autonomia.

JATLite no dona especificacions per la creació d'un agent pròpiament dit. És a dir, no implementa de cap manera una classe *agent*, que nosaltres poguéssim implementar a l'aplicació, etc. Dóna eines per fer-ho. A més de conèixer la semàntica de les funcions de JATLite, la persona que dissenyi una aplicació ha de tenir clar allò que necessita, i crear-ho.

En aquest aspecte, és important tenir coneixements previs de les propietats/atributs que defineixen els agents, per agafar-los com a model de la nostra implementació. Així doncs JATLite no construeix per ell mateix agents intel·ligents que busquen informació o que automatitzen tasques humanes. L'investigador és lliure d'usar teories o tècniques més convenients per a l'aplicació o recerca. JATLite no facilita l'ús de cap eina en particular de raonament.

Un exemple és JESS. És un sistema expert que ha estat embolcallat per JATLite per a produir agents intel·ligents basats en Java i regles.

9.3. ARQUITECTURA DE JATLITE

L'arquitectura de JATLite està formada per cinc nivells: *protocol*, *router*, *KQML*, *Base* i *Abstract*. (v. Figura 13).

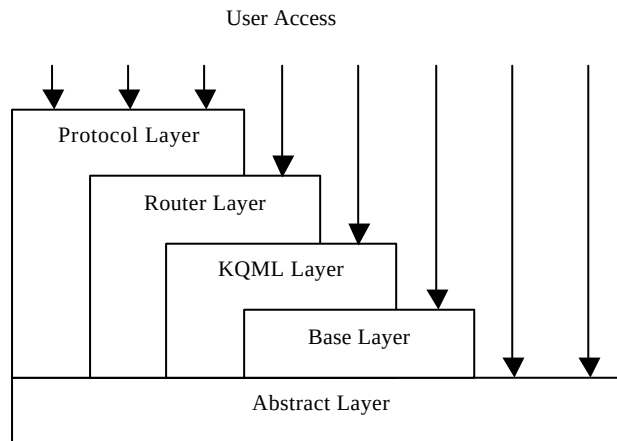


Figura 8.2: Nivells de l'arquitectura JATLite.

En la Figura 8.2 es poden veure els nivells que defineix JATLite. Podem veure com l'usuari pot accedir directament a tots els nivells indistintament. Això permet a un usuari treballar amb tot el sistema. Es té una jerarquia de nivells especialitzada segons les funcions que s'hi realitzen, però tots els nivells accedeixen al nivell més baix perquè aquest nivell conté tota la informació que cal per realitzar aquestes funcions.

9.3.1.1. Abstract layer

No és estrictament un nivell sinó que dona les classes per implementar els nivells superiors. Aquest nivell inclou nou classes, i diferents estructures de dades (v. Taula 9.1).

<i>Address</i>	Conté la informació necessària per establir una connexió entre agents. Cada registre defineix el tipus de connexió, host i port.
<i>AddressTable</i>	Cada agent té informació de tots els agents a qui pot adreçar-se. Quan s'inicialitza un agent cal donar aquesta informació inicial. Després podrem afegir/treure connexions si és necessari. L'ANS és distribuït a cada agent.
<i>AgentAction</i>	AgentAction dona facilitats pel manegament dels missatges rebuts. Bàsicament aquesta classe ens permetrà de llegir un missatge quan aquest arribi.
<i>Connection</i>	Dóna les característiques de la connexió entre agents. Fixa el protocol de transmissió a TCP/IP, encara que el programador pot canviar-lo a UDP.
<i>ConnectionTable</i>	És una taula que conté els noms dels agents i si es pot rebre o no. En cas de voler obrir més d'una connexió amb un agent, cal donar noms diferents a les connexions. Si obrim dues connexions amb el mateix nom de recepció, la primera s'atura i només quedarà escoltant la segona.
<i>MessageBuffer</i>	És la cua de missatges per enviar o rebuts.
<i>ReceiverThread</i>	És un thread obert sobre un socket que està escoltant. Quan rep un missatge s'activarà la classe AgentAction. Permet d'enviar/rebre cadenes de caràcters.
<i>Security</i>	La seguretat és a nivell de connexió. Assegura que totes les connexions que s'estableixin siguin segures.
<i>ServerThread</i>	Abans d'inicialitzar un ReceiverThread cal acceptar la connexió. El ServerThread està escoltant el medi i és qui fa l'accept a una nova connexió.

Taula 9.1: Classes definides per *Abstract Layer*.

Com veiem aquest nivell és bàsic pel desenvolupament de qualsevol funció dels nivells superiors.

La principal característica és la implementació de l'*AddressTable* i *ConnectionTable*, que descentralitza l'ANS. Ara bé, cal inicialitzar totes les connexions que es vulguin fer una a una, a cadascun dels agents. Com veurem, quan s'inicialitza l'agent, podem fer que es llegeixi un script amb les adreces dels agents del sistema, i automàticament es construeix l'*AddressTable* inicial.

9.3.1.2. Base layer

Aquest nivell proporciona eines de comunicació (enviar/rebre missatges) basades en TCP/IP.

El *base layer* és el primer dels nivells de l'arquitectura JATLite i implementa les classes d'*AbstractLayer*. Així que bàsicament les classes que s'implementen són les mateixes.

- *BAddressTable* és la implementació d'*AbstractLayer.AddressTable*. Manega la informació relativa a les adreces dels agents del sistema. Podem inicialitzar-la estàticament llegint un fitxer, o dinàmicament usant els mètodes que ho permeten.
- *BConnectionTable* manté la taula de connexions obertes. Cal notar que les connexions s'inicialitzen/mantenen usant sockets. Aquest fet és important perquè afecta tant a l'eficiència com a les dades que es poden transmetre. Podrem enviar/rebre caràcters o cadenes de caràcters.
- *BAgentAction* és una implementació d'*Abstract.AgentAction*. Dóna l'entorn per manegar els missatges rebuts. Quan arribi un nou missatge aquest s'afegirà al buffer de nous missatges, i podrà ser llegit pel receptor.
- La seguretat (*BSecurity*) chequeja la validesa dels noms dels agents abans de donar per oberta una nova connexió.
- Cal veure que tot i que s'encapsulen els missatges per enviar, i els rebuts, en una classe *Object*, en última instància aquests missatges són transmesos/guardats en forma de caràcters (*String*).
- *BServerThread* és un thread que accepta (*accept()*) connexions d'altres agents. Quan n'accepta una genera un *BRecvThread* per escoltar i enviar missatges. Pel protocol de connexió, el *Base Layer* assumeix que quan un agent espera connectar amb un altre agent, inicia la connexió i espera la confirmació (*connected*). Llavors, si això té èxit, s'intercanvien els noms dels agents.

Si no es volen implementar aquestes característiques, hom ha de reescriure els mètodes que facin falta per tal d'aconseguir-ho.

9.3.1.3. KQML Layer

És el nivell immediatament superior al *Base Layer*, i les classes de *KQMLLayer* són subclasses d'aquest.

Aquest nivell restringeix que el pas de missatges entre agents ha d'usar el llenguatge de comunicació entre agents KQML.

- Les classes que proporciona són subclasses de *BaseLayer*. No s'han implementat totes doncs no cal mantenir control de les connexions ni les adreces. Només s'han implementat: *KQMLAgentAction*, *KQMLServerThread*, *KQMLSecurity*, i *KQMLRecvThread*.
- A més es proporciona una nova classe (amb llurs mètodes) que inclou les funcions dels missatges en KQML. Permet crear un nou missatge, consultar/assignar els diferents paràmetres.
- En aquests missatges el format normalment és de llistes (v. KIF), per això s'han implementat algunes funcions per anar recorrent aquestes llistes.

9.3.1.4. Router Layer

Característiques:

- El Router és una aplicació especialitzada que rep missatges d'agents registrats i els *encamina* al destinatari correcte. Aquest missatge serà encuat a la cua de missatges rebuts del destinatari.
- Per encaminar els missatges, implementa l'*Agent Name Service* (ANS) que té coneixement de totes les adreces i connexions obertes.
- Registres. El Router dóna facilitats per tal de registrar-se per tenir dret a usar-lo.
- Llista d'agents. Un agent pot enviar una llista d'agents al Router per tal d'esbrinar el seu estat actual.
- Desconnexió. Un agent pot enviar un missatge de "disconnect-agent" al Router si no vol rebre més missatges. Si un agent es desconnecta accidentalment, el Router detecta un temps d'inactivitat, i si després d'intentar connectar-hi no hi ha èxit, s'esborra el registre de l'agent.
- Reserva de missatges. Podem reservar la recepció de missatges.

9.3.1.5. Protocol Layer

El *Protocol layer* és el nivell més alt. Aquest nivell suporta diversos serveis estàndards d'Internet com ara SMTP (*Simple Mail Transfer Protocol*) i FTP (*File Transfer Protocol*).

Per utilitzar aquests dos serveis, JATLite ha dissenyat algunes funcions.

Permet l'ús d'aquestes eines de forma segura, demanant login i password per tal d'establir alguna connexió a través d'Internet.

9.4. DESENVOLUPAMENT D'UNA APLICACIÓ JATLITE

L'aplicació en qüestió és: *el món de la brossa*.

En aquesta aplicació hi ha una zona (àrea) de treball on, periòdicament es genera brossa. Llavors, en aquest entorn hi ha uns elements que estan pendents d'allò que passa, i quan hi ha una brossa per recollir, van de la posició actual a la de la brossa i la recullen.

En aquest sistema hi ha dos tipus d'agent: el *món*, i els *robots*.

- El món és l'encarregat de generar les brosses periòdicament. Informa de la seva presència al/s robot/s actius i aquest van a recollir-la.
- El robot és un element autònom (està escoltant tota l'estona) i quan rep l'ordre d'una nova brossa, hi va, i la recull.

En aquest sistema, si hi ha més d'un robot, per no xafar-se els uns amb els altres, quan un robot recull una brossa, notifica el fet a tots els agents del sistema. Al món no li afecta aquesta informació, però si la rep un altre robot, aquest s'atura a l'instant.

Com es diu a la descripció, JATLite no proporciona cap interfície gràfica estàndard que ajudi al desenvolupament d'una aplicació, per això s'ha aprofitat una interfície que es dona en un dels exemples de demostració (v. Figura 9.3). Aquesta interfície, inicialment implementava el diàleg entre dos agents usant pas de missatges KQML; hi ha facilitats per establir connexions, compondre i enviar nous missatges.

Bé, aquesta interfície com a base està molt bé perquè hi ha tot allò (i més) que necessitem per desenvolupar aquesta aplicació. Té dues àrees destinades als missatges rebuts i enviats respectivament. Hi ha botons per connectar, compondre un nou missatge, acabar, i una llista desplegable que permet fàcilment de dir amb quin/s agent/s ja registrats vols connectar.

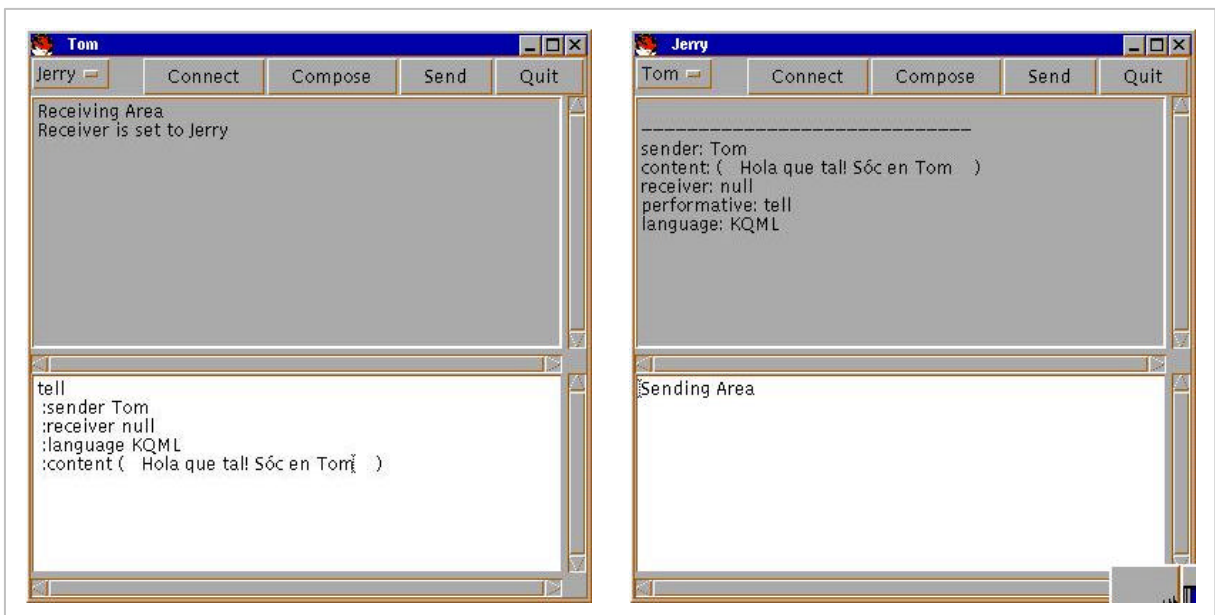


Figura 9.3: Interfície original utilitzada. Veiem com es fa el pas d'un missatge de l'agent Tom a l'agent Jerry, i les opcions per manegar missatges.

9.4.1. Descripció del desenvolupament

JATLite com s'ha vist, *només* proporciona unes eines per ajudar al desenvolupament de SMA. Aquest és un fet molt important, doncs no implementa una visió general d'agent. És a dir, com veurem, la construcció general de l'agent l'hem de realitzar nosaltres. Així s'ha creat un *thread*⁹ per tal de monitoritzar el sistema i crear un entorn multiagent.

En el nostre sistema es tenen dos tipus d'agents: el món i els robots. El món monitoritza l'execució del sistema i va generant periòdicament una brossa en una posició aleatòria. Els robots, que estan esperant, quan detecten aquesta nova brossa, hi van des d'on siguin.

Els noms dels agents són: *Silvestre* (món), *R2D2* (robot1) i *Piolin* (robot2).

9.4.1.1. Connexions: medi de transmissió de missatges

Les operacions que dona JATLite són bàsicament de comunicació.

En la demostració, tots els agents són a la mateixa màquina; cal només establir els ports per cadascuna de les connexions. Es pot implementar aquesta aplicació en varies màquines; JATLite no permet d'especificar adreces IP directament, només el *host name*.

Ja s'ha dit que les connexions es poden realitzar d'una manera estàtica (quan s'inicialitza el sistema), o de manera dinàmica (intercanviant un diàleg amb l'usuari).

En l'aplicació s'ha escollit la forma estàtica. Per realitzar aquesta funció s'han escrit uns *scripts* que el sistema llegeix a l'inici i crea l'*AddressTable* i la *ConnectionTable*.

Cada registre d'aquests *scripts* té un format determinat, però JATLite n'informa del seu correcte ús.

En la Figura 9.4 podem veure els *scripts* d'inicialització de connexions, i la seva interpretació física.

Cada registre d'aquests *scripts* té un format determinat i permet establir la cada connexió, per exemple, si tenim:

```
Piolin,null,5555,BServerThread,(ocell)
Silvestre,localhost.localdomain,4444,BRecvThread,(gat)
```

ens diu que Piolin establirà una “porta” per enviar dades, i una porta per escoltar les dades que li envii Silvestre.

⁹ Els *threads* són fils d'execució que implementa Java, i cada fil estableix un camí d'execució independent.

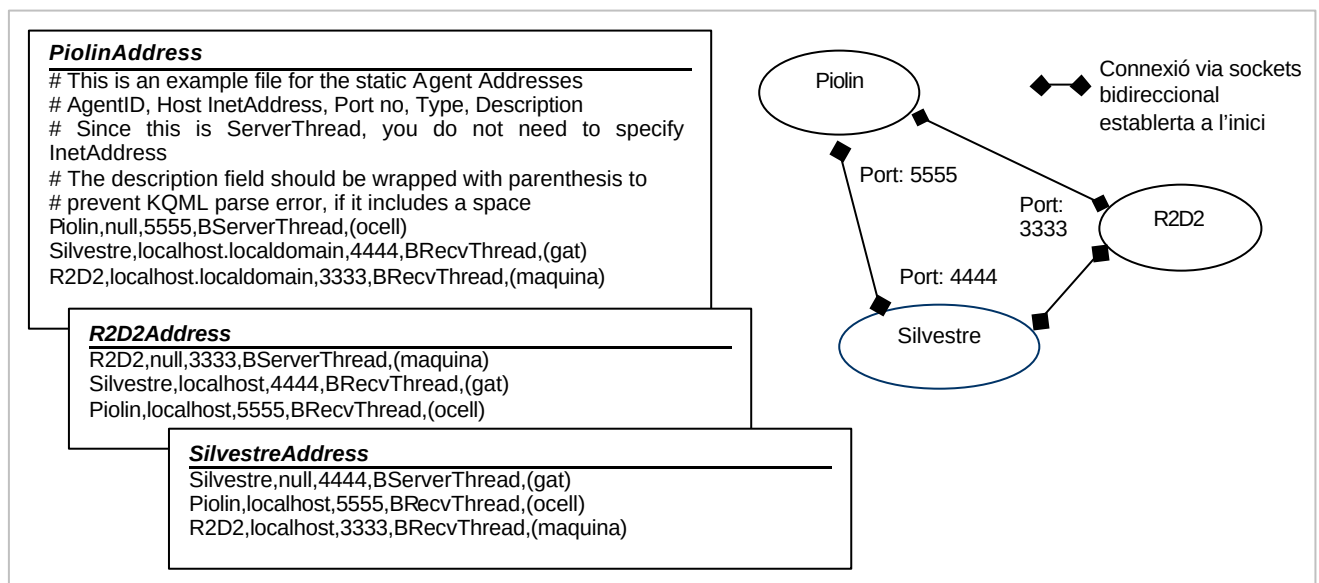


Figura 9.4: Connexions establertes entre els agents del sistema.

Aquests *scripts* es “passen” com a paràmetre a l’inicialitzar els agents:

```
java rubbish.exemple.SASyncExample Piolin rubbish/PiolinAddress
```

L'altra forma de connectar els agents és engegar un nou agent i, si es vol una connexió, es pot implementar algun tipus d'interfície amb tots els paràmetres que es necessiten (v. Figura 9.5). En ambdós casos cal tenir coneixement dels ports disponibles en cada màquina per establir les connexions.

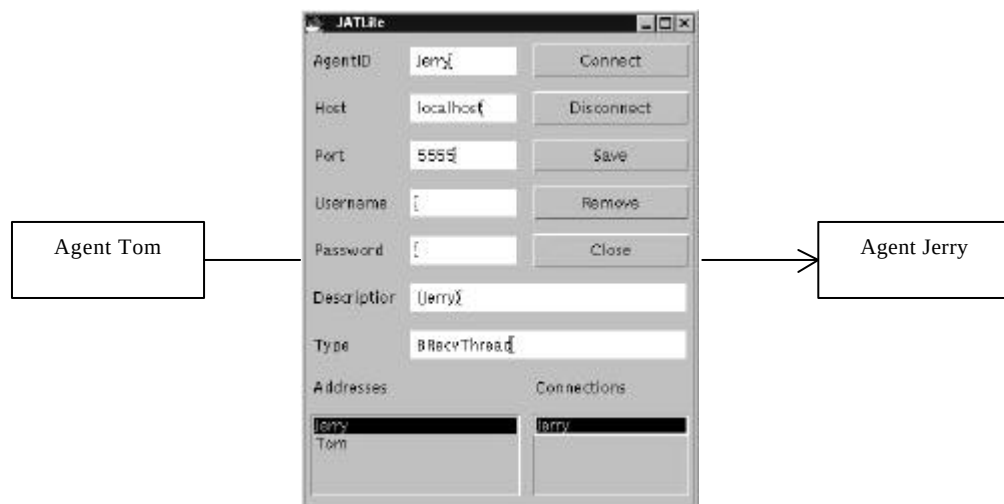


Figura 9.5. Connexió establerta dinàmicament. L'agent Tom vol establir una connexió amb l'agent Jerry a través del port 5555 d'aquest últim.

En el fragment següent podem veure com s'inicialitzen les taules de connexions i adreces entre els diferents agents.

Com hom pot veure als *scripts* d'inicialització, el programador sap els ports per on es comunicaran els agents.

El fragment següent del codi de l'aplicació mostra com inicialitzem les connexions:

// File: /mnt/d/pfc/JATLite/jatlitelinux/rubbish/exemple/SASyncExample.java

```

...
while(readline != null) {
    if(!readline.startsWith("#")) {
        Address addr = new Address(readline);
        _addresses.addAddress(addr);

        String type = addr.getType();
        String name = addr.getID();
        String descr = addr.getDescription();

        if(type.equals("BServerThread")) {
            ServerThread server =
            createServerThread(name,Thread.NORM_PRIORITY);
            if(server == null) {
                String args[] = new String[1];
                args[0] = addr.getID();
                processMessage("InitializeFailed",args);
            }
            System.out.println("Creat un ServerThread "+getName());
        }
        else
            if(type.equals("BRecvThread")) {
                try {
                    createReceiverThread(name,Thread.NORM_PRIORITY);
                } catch (Exception e) {
                    // this error may happen if the server is not
                    // Ignore.
                }
                System.out.println("Creat un ReceiverThread "+getName());
            }
            else {
                String args[] = new String[1];
                args[0] = type;
                processMessage("UnsupportedType",args);
            }

        if (var_ctrl==1) {
            if (descr.equals("(gat)")) {
                _agentFrame.setClient();
                typeAgent=1;
            }
            else
            {
                _agentFrame.setServidor();
                typeAgent=2;
            }
        }
        var_ctrl++;
    }
    readline = input.readLine();
    //System.out.println("Linia llegida:"+readline);
}
...

```

Inicialitzem una nova taula d'adreces. Per qüestions internes de JATLite s'ha d'afegir un primer registre *sempre*. Ens hem d'habituar a treballar sobre variables definides per JATLite com `_addresses`.

Podem consultar tots els paràmetres de les adreces.

En el cas de tenir una adreça de tipus *BServerThread*, creem un thread del tipus *KQMLServerThread*, que acceptarà connexions d'altres agents; si es tracta de *BRecvThread*, instanciem la classe per tal de poder manejar els missatges que arribin a l'adreça que s'inicialitza.

Dins la implementació del codi cal diferenciar de quin tipus d'agent es tracta: si l'agent és el gat, aquest serà el món; sinó iniciarem la interfície i implementació dels robots (v. paràgraf següent)

Si es volen crear dinàmicament, podem fer servir la interfície de la Figura 9.5. Aquesta interfície no està implementada com una classe que l'usuari pot cridar i sortir-ne amb una nova connexió. Ara bé, en un exemple que ens proporciona JATLite, està implementada en dos fitxers Java (*SASyncConnectionDlg.java* i *SASyncConnectionTable.java*).

Si cal, el programador pot reformar-la, però està prou bé i és fàcil d'usar. No s'ha inclòs el codi d'aquesta interfície dins el codi general perquè es pot utilitzar sense més.

Fragment de *SASyncFrameExample.java* (interfície de l'agent on anem veient les evolucions) on es maneguen les connexions actuals/noves.

```
public void actionPerformed(ActionEvent evt) {
    String arg = evt.getActionCommand();

    System.out.println("Event Activat: "+evt.toString());
    System.out.println("Accio: "+arg);
    if("Connect".equals(arg)) {
        SASyncConnectionDlg connectionDlg = new SASyncConnectionDlg(this,
        _agentAction.getAddressTable(),_agentAction.getConnectionTable());
        connectionDlg.showDlg();
    }
    else
        if("Compose".equals(arg)) {
            ...
        }
}
```

La interfície de diàleg amb l'usuari rep com a paràmetres l'entorn de l'agent, les adresses dels agents, i el nom de les connexions obertes.

9.4.1.2. Creació/funcionament dels agents

En el nostre cas, tots els agents es creen amb el mateix fitxer.

Per crear els diferents tipus d'agents s'utilitza el camp *descripció* dels *scripts* d'inicialització. Es mira el valor d'aquest camp i si és un gat (Silvestre), es crearà l'agent client (món), i en cas contrari es crearan els servidors (v. paràgraf anterior).

JATLite deixa a l'elecció del programador la forma d'implementar el sistema. Es podia haver pensat en implementar els agents com *threads* amb independència els uns dels altres, i en fitxers diferents. Sí, és una possible millora, que podria tractar-se en una versió posterior d'aquesta aplicació, una vegada après el funcionament/possibilitats que JATLite dóna. En el disseny d'aquesta aplicació s'ha pensat en un únic fitxer, ara bé cada agent continua essent un *thread* que funciona independentment.

Per la nostra aplicació el gestor dels agents és un *thread* anomenat **monitor**, que presta uns serveis o altres depenent del tipus d'agent on siguem.

Veiem la part més important del funcionament dels agents; és el fragment de *SASyncExample.java* on es gestiona el *thread* monitor:

```
//*****
class monitor implements Runnable {
    Thread t;
    int typeAgent=-1;
    int num_iteracions=15;
    SASyncFrameExample frameExemple;
    String nomFrame;

    monitor(int tipusAgent,Frame context,String nom) {
        typeAgent=tipusAgent;
        nomFrame=nom;
        frameExemple=(SASyncFrameExample)context;
        t=new Thread(this,"Monitor");
        System.out.println("Estem al fil: "+t.getName());
        t.start();
    }

    //Codi del fil monitor
    public void run() {
        if (typeAgent == 1) {
            for(int i=num_iteracions; i>0; i--) {
                try {
                    System.out.println("Vaig a dormir");
                    Thread.sleep(15000);
                    System.out.println("Hola, ja estic despert!");
                    System.out.println("Queden "+i+" iteracions");
                    frameExemple.creaMissatgeBrossa();
                } catch (InterruptedException e) { }
            }
        }
        if (typeAgent == 2) {
            System.out.println("Soc el THREAD del Servidor");
            frameExemple.iniPosicio();
            posiciones _f = new posiciones(nomFrame);
            _f.start();
            int x,y;

            _f.pinta(2,frameExemple.getRX(),frameExemple.getRY());
            _f.repaint();
            while(true) {
                try {
                    t.sleep(300);
                    if (frameExemple.HiHaRubbish()) {
                        int varctrl=-1;
                        x=frameExemple.getX();
                        y=frameExemple.getY();
                        _f.pinta2(frameExemple.getRX(),frameExemple.getRY(),x,y);
                        _f.repaint();
                        do {
                            varctrl=frameExemple.recollirRubbish(x,y,_f);
                            t.sleep(50);
                        } while ((varctrl!=1) && (varctrl!=2));
                        if (varctrl==1) {
                            System.out.println("MOLT BE, BROSSA RECOLLIDA");
                            _f.pinta(3,x,y);
                            _f.repaint();
                        }
                        else if (varctrl==2) {
                            System.out.println("BROSSA RECOLLIDA... PER ALGU
                            ALTRE");
                        }
                    }
                } catch (InterruptedException e) { }
            }
        }
    }
}
```

El monitor, que regula el comportament dels agents és un *thread* o fil d'execució.

Quan s'inicialitzen les connexions, s'inicialitza aquesta variable. Si el valor és 1, es tracta del Silvestre. En aquest cas, el codi simula la generació de brosses cada 15 segons, fins a un màxim de 15 (és una decisió de disseny, totalment subjectiva, doncs només es tracta d'una simulació)

Si el valor de *typeAgent* és 2, es tracta d'un robot. Primer s'inicialitza aleatòriament la seva posició inicial dins l'àrea permesa. La missió d'un robot és estar pendent de l'arribada d'una brossa. Quan n'hi ha, cal anar a buscar-la. Per simular el moviment del robot, quan treballa per anar a recollir la brossa, la seva posició es va actualitzant cada 50ms.

_f és el tauler on es veuen les evolucions dels robots, anant cap a la brossa

Recordem que un agent és un *thread*. En aquest cas sempre està escoltant. Si no es fica aquest *sleep*, aquest agent no deixaria treballar a la resta. S'ha fixat que cada 0.3s mira si hi ha o no una nova brossa.

Fem un moviment cap a l'objectiu de la brossa.

Pot passar que l'aconseguim recollir (*var_ctrl==1*) nosaltres, o que ens la prenguin abans (*var_ctrl==2*).

9.4.1.3. Comunicació

Com s'ha vist en l'apartat que descriu l'arquitectura de JATLite, hi ha un nivell *dedicat* a la definició d'un entorn per entendre's amb el llenguatge KQML (v. p.3.2.2.).

Emissor

Veiem com es crea un missatge en JATLite:

```
public void construirMissatgeBrossaRecollida(int xr, int yr) {
```

```
    String Message = "recollida \r\n";
    Message = Message + " :sender " + getTitle() + " \r\n";
    Message = Message + " :receiver All \r\n";
    Message = Message + " :language KQML \r\n";
    Message = Message + " :content "+xr+", "+yr;
```

```
    try {
```

```
        _sendArea.setText(Message);
```

```
        String sendstr=_sendArea.getText();
```

```
        String msg="(";
        msg=msg+filtering(sendstr)+")";
```

```
        _agentAction.broadcast(msg);
```

```
        System.out.println("El missatge filtrat es: ");
        System.out.println(msg);
```

```
    } catch (ConnectionException ce) {
        _sendArea.append(ce.toString());
    }
```

```
}
```

El missatge que s'envia ha d'estar lliure de caràcters especials (<EOL> i <CR>)

El format del missatge és:

tipus_acció

:sender nom_agent_origen

:receiver nom_agent_desti

:language llenguatge_del_contingut

:content contingut

Podem incloure la resta de camps que KQML defineix en la seva sintaxi: *in-reply-to*, *force*, *ontology*, *reply-with*. En aquest no cal perquè fem servir un *tell* normal.

La transmissió pot ser directa d'agent-origen a agent-destí, o fent broadcast als agents que hagi actius en aquell moment..

Receptor

Ara veurem com es tracta al receptor. Com ja s'ha dit, el missatge es guarda en el *MessageBuffer* del receptor però el *RecvThread* avisa a l'agent d'una nova recepció. Aquest avís activa un mètode implementat en *AgentAction*: *Act(object o)*, on *o* és el missatge.

```
protected boolean Act(Object o) {
```

```
    try {
```

```
        KQMLMessage kqml = new KQMLMessage((String)o);
        String perf = kqml.getValue("performative");
        String position = kqml.getValue("content");
```

```
        System.out.println("El valor de perf es: "+perf);
        System.out.println("El valor de content es: "+position);
        System.out.println("Missatge Rebut:");
        System.out.println(" "+kqml.toString());
```

```
        if (perf.equals("brossa")){_agentFrame.setRubbish(position);}
```

```
        if (perf.equals("recollida") && typeAgent==2){
            _agentFrame.aturarRecollida();
```

```
            _agentFrame.processMessage(kqml);
```

```
            if(perf.equals("disconnected")) {
```

```
                String args[] = new String[1];
                args[0] = kqml.getValue("sender");
                _connections.removeConnection(args[0]);
                _agentFrame.processMessage("RemoveConnection", args);
            }
```

```
    } catch (ParseException e) {
```

```
    }
```

```
    return true;
```

```
}
```

Tenim dos tipus de performatives pròpies del nostre domini: *brossa* i *recollida*. La primera avisa d'una nova *brossa*; la segona avisa que una *brossa* ha estat recollida per algú altre.

kqml és el missatge entrant. Podem fer la consulta de qualsevol paràmetre que ens interressi, i tractar el tipus d'acció.

Ensenyem el missatge rebut a la interfície de l'agent.

Tractem el cas de rebre una performative de desconnexió d'algun agent. Llavors cal esborrar de la nostra "agenda" aquesta connexió.

9.4.1.4. Tipus d'accions de comunicació o performatives

En la nostra aplicació fem servir el llenguatge de comunicació KQML. KQML defineix un conjunt de *performatives* que donen resposta a la gran majoria de problemes.

Unes de les ventatges que proporciona KQML és l'adaptabilitat, és a dir, podem crear nous tipus sense problemes, i gestionar-les a les nostres aplicacions.

En el nostre cas, hem creat dos *performatives* especials pel nostre domini d'aplicació: *brossa* i *recollida*. La primera l'envia el món als robots i els hi diu que ha creat una nova brossa i en dóna la posició; la segona, l'envia el robot que ha recollit una brossa, i indica als altres robots actius al sistema que es poden aturar.

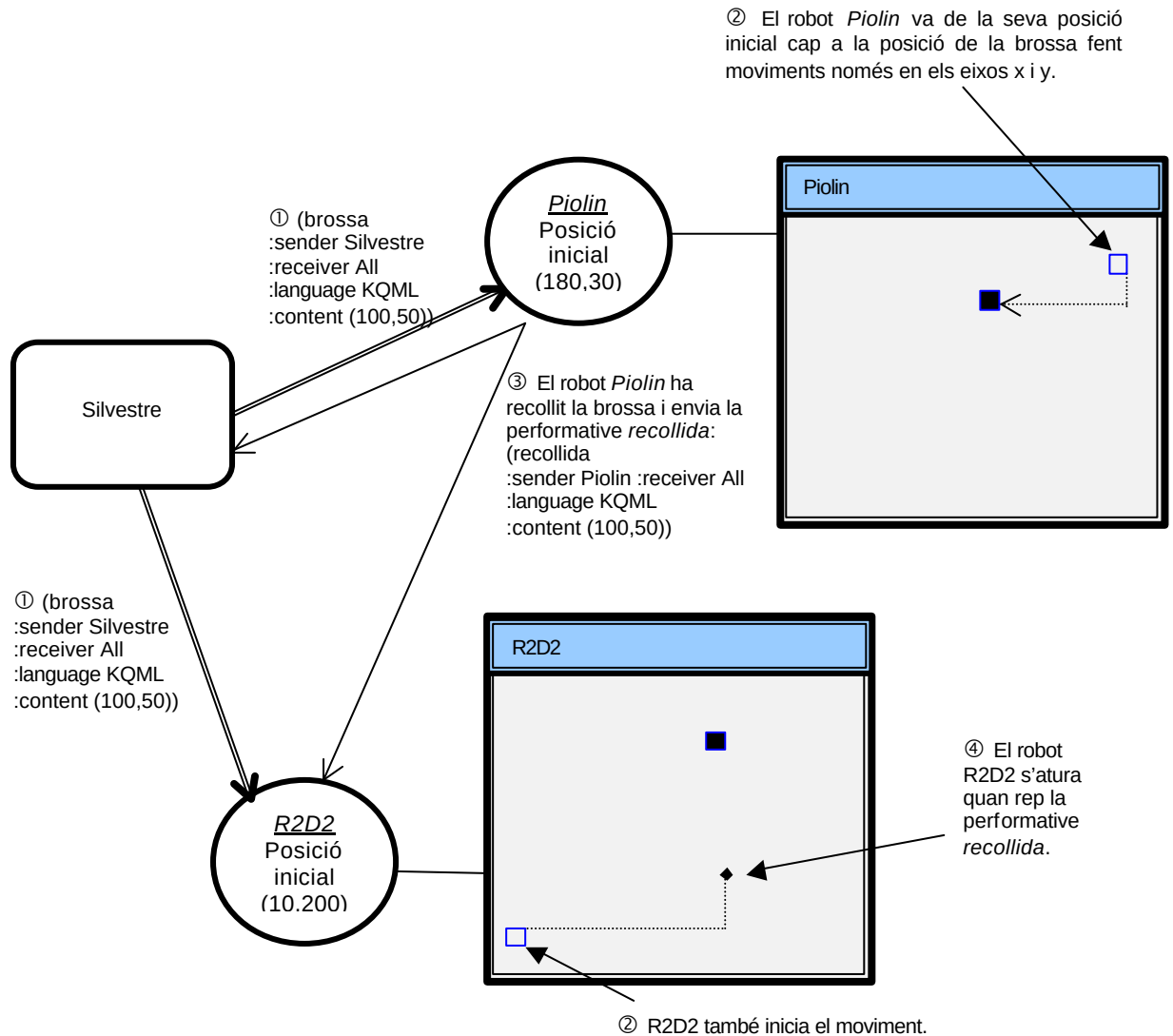
A més d'aquestes dues accions, el *router layer* defineix un altre conjunt que no són compatibles amb la definició de l'estàndard KQML (v. Taula 9.2)

Connexió	Reconnect-agent	Un agent registrat pot intentar tornar a connectar amb l'agent ex: (reconnect-agent :sender me :receiver Router :password xxxx :host java.sun.com port 1234) (el port, host i passw són opcionals)
	disconnect	Desconnectem la connexió ex: (disconnect :sender me :receiver Router)
Registre	Register	Un agent vol registrar-se al seu Router ex: (register :sender Tom :receiver Router :password xxxx)
	identify	El router pregunta sobre la identitat de l'agent amb qui vol connectar. ex: (identify :sender Router :receiver: Tom)
	whoiam	L'agent es presenta. Dona la seva localització per tal que altres agents puguin posar-s'hi en contacte. Es dóna informació del host i el port, però es pot donar adreces de correu electrònic, de planes web, etc. ex: (whoiam :sender Tom :receiver Router :message-method MessageRouter :content (contact-information :host cdr.stanford.edu :port 1234 :email jhc@cdr.stanford.edu) :KQML-extensions http://java.sun.com/newProtocol :description (advertize :type DesignAgent))
	unregister	Un agent es dóna de baixa del Router. ex: (unregister :sender Tom :receiver Router :password xxxx)
Llistes d'agents	list-users	Preguntem informació sobre la llista d'agents actual. ex: (list-users :sender Tom :receiver Router)
	registered-agent	Contesta del Router de list-users ex: (registered-agents :sender Router :receiver Tom :content ((he java.sun.com connected)(she aaa.bbb.com disconnected)))
Adreces	request-address	Pregunta l'adreça d'un agent al Router. ex: (request-address :sender Tom :receiver Router :content him)
	agent-address	Resposta del Router al request-address ex: (address :sender Router :receiver Tom :name he :host cdr.stanford.edu :port 1234 :description (agent-info :service xxxx))
Reserva mis.	reserve-message	Donem una petició de reserva de recepció de missatges al Router. ex: (reserve-message :sender Tom :receiver Router :content (1 3 5) :time 819374635243 :host cdr.stanford.edu :port 1234)
Borrar mis.	delete-message	Un agent vol esborrar un missatge de la seva bustia. ex: (delete-message :sender Tom :receiver Router :content 3)

Taula 9.2: Accions de comunicació o performatives definides pel Router de JATLite.

9.4.1.5. Esquema del sistema

En la Figura 9.6 podem veure les diferents interfícies del sistema; en l'esquema següent podem veure els passos, i les comunicacions que s'estableixen entre tots els elements del sistema.



La freqüència de generació és de 15 segons. Si en aquest temps el robot no ha tingut temps per arribar a la brossa, no passa res. Quan un robot rep una brossa, aquest es dedica només a fer això. Si arriben missatges d'aparició de noves brosses mentrestant, es van guardant en la cua de missatges rebuts de l'agent. Quan s'ha recollit la brossa o s'atura perquè un altre robot l'ha recollit abans, es detecta que hi ha missatges a la cua, es llegeix i es continua treballant; no es perd res.

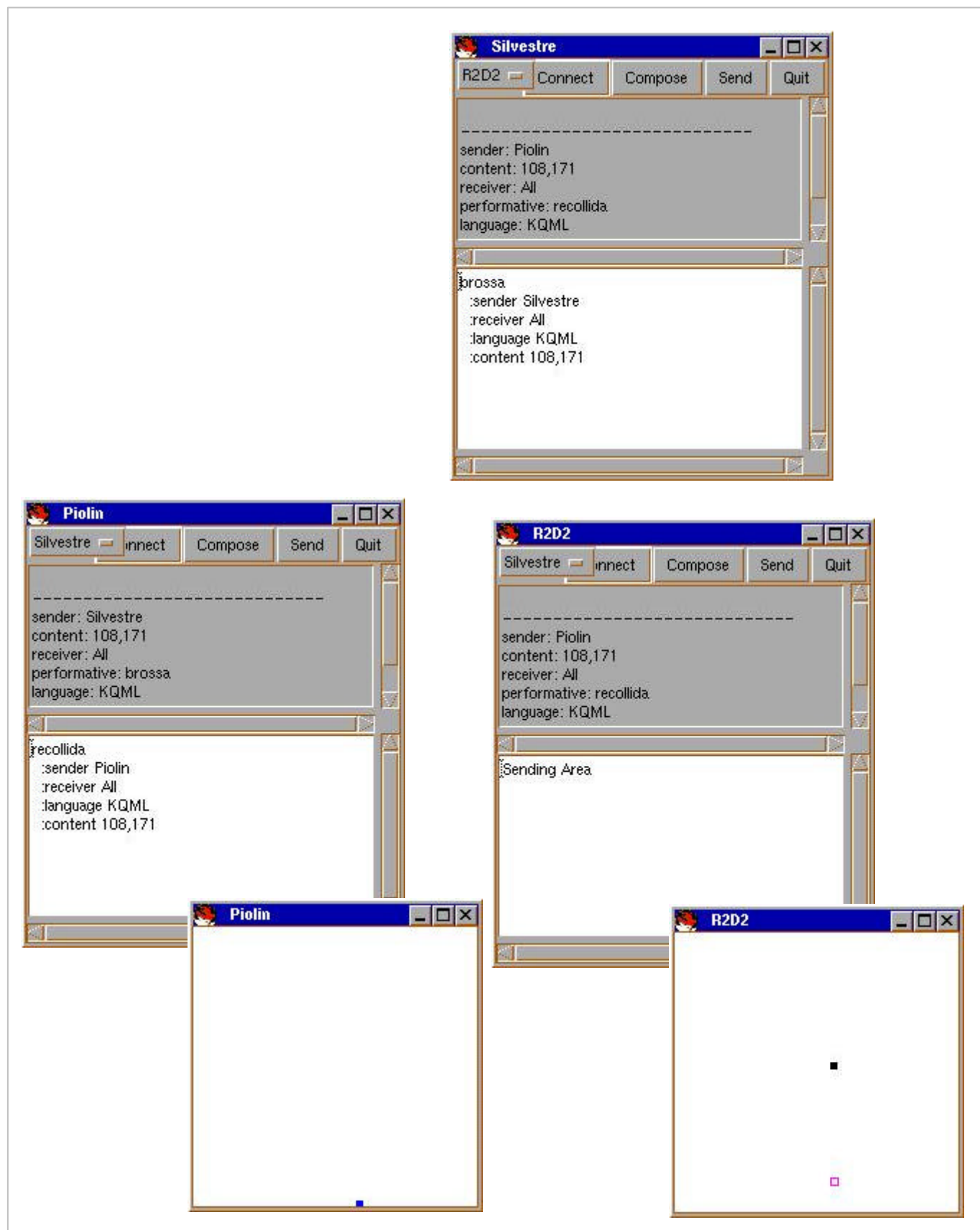


Figura 9.6: Interfícies del sistema. A dalt, la interfície del món (Silvestre). A l'esquerra, la interfície de Piolin i el seu tauler on veiem les evolucions. A baix a la dreta, veiem la interfície de l'altre robot, R2D2, amb el seu tauler.

9.4.2. Codi font de l'aplicació

9.4.2.1. Cos principal

```
//*****
// File: /mnt/d/pfc/JATLite/jatlitelinux/rubbish/exemple/SASyncExample.java

package rubbish.exemple;

import java.applet.Applet;
import java.lang.*;
import java.io.*;
import java.util.*;

import Abstract.*;
import BaseLayer.*;
import KQMLLayer.*;

import java.awt.*;

public class SASyncExample extends KQMLAgentAction {

    monitor m;
    int typeAgent=-1;
    SASyncFrameExample _agentFrame;
    public void setAgentFrame(SASyncFrameExample frame) {
        _agentFrame = frame;
        ((SASyncConnectionTable)_connections).setAgentFrame(frame);
    }

    public SASyncExample() {
        super();
    }

    public SASyncExample(String id, String addressFilePath) {
        super(id);
        try {
            _addresses = new BAddressTable();
            _addresses.addAddress(new Address(id,null,-1,"BRecvThread","Client"));
            initDataMembers();

            // KQMLAgentFrame
            SASyncFrameExample kqmlframe = new SASyncFrameExample(id,this);
            setAgentFrame(kqmlframe);
            kqmlframe.show();

            BufferedReader input = new BufferedReader(new FileReader(new
            File(addressFilePath)));
            String readline = input.readLine();
            int var_ctrl=1;

            //System.out.println("Linia llegida: "+readline);
            while(readline != null) {
                if(!readline.startsWith("#")) {
                    Address addr = new Address(readline);
                    _addresses.addAddress(addr);
                    String type = addr.getType();
                    String name = addr.getID();
                    String descr = addr.getDescription();

                    System.out.println("Descripcio: "+descr+" i iteracio: "+var_ctrl);

                    if(type.equals("BServerThread")) {
                        ServerThread server =
                        createServerThread(name,Thread.NORM_PRIORITY);
                        if(server == null) {
                            String args[] = new String[1];
                            args[0] = addr.getID();
                            processMessage("InitializeFailed",args);
                        }
                    }
                }
            }
        }
    }
}
```

```

        System.out.println("Creat un ServerThread "+getName());
    }
    else
        if(type.equals("BRecvThread")) {
            try {
                createReceiverThread(name,Thread.NORM_PRIORITY);
            } catch (Exception e) {
                // this error may happen if the server is not
                // Ignore.
            }
            System.out.println("Creat un ReceiverThread "+getName());
        }
        else {
            String args[] = new String[1];
            args[0] = type;
            processMessage("UnsupportedType",args);
        }

    if (var_ctrl==1) {
        if (descr.equals("(gat)")) {
            _agentFrame.setClient();
            typeAgent=1;
        }
        else
        {
            _agentFrame.setServidor();
            typeAgent=2;
        }
    }
    var_ctrl++;

    }
    readline = input.readLine();
    //System.out.println("Linia llegida:"+readline);
}

input.close();
System.out.println("Inicialitzacio de fitxer acabada");
m=new monitor(typeAgent,_agentFrame,id);

System.out.println("Hem creat el monitor");

//t = new Thread("Monitor");
//t.setPriority(Thread.MIN_PRIORITY);
//t.start();
//System.out.println("Thread monitor creat");

} catch (Exception ee) {
    // critical error, can't start
    String args[] = new String[1];
    args[0] = ee.toString();
    processMessage("InitializeFailed",args);
}
}

protected void initDataMembers() {
    // address table _addresses should not be null
    _security = new KQMLSecurity(_addresses);
    _connections = new SASyncConnectionTable();
    _queue = new BMessageBuffer();
    _endWith = '\004';
    _durationTime = -1;
}

public void processMessage(String command,Object obj) {
    String[] args = (String[])obj;

    if(command.equals("UnsupportedType")) {
        System.out.println(command + " " + args[0]);
    }
    else

```

```

        if(command.equals("InitializeFailed")) {
            System.out.println(command + " " + args[0]);
            endAction();
        }
        // invoked from _agentFrame
        else
            if(command.equals("Quit")) {
                endAction();
            }
    }

protected boolean Act(Object o) {
    try {
        KQMLmessage kqml = new KQMLmessage((String)o);
        String perf = kqml.getValue("performative");
        String position = kqml.getValue("content");

        System.out.println("El valor de perf es: "+perf);
        System.out.println("El valor de content es: "+position);
        System.out.println("Missatge Rebut:");
        System.out.println("  "+kqml.toString());

        if (perf.equals("brossa"))
        {
            _agentFrame.setRubbish(position);
        }

        if (perf.equals("recollida") && typeAgent==2)
        {
            _agentFrame.aturarRecollida();
        }

        _agentFrame.processMessage(kqml);
        if(perf.equals("disconnected")) {
            String args[] = new String[1];
            args[0] = kqml.getValue("sender");
            _connections.removeConnection(args[0]);
            _agentFrame.processMessage("RemoveConnection",args);
        }
    } catch (ParseException e) {
    }
    return true;
}

public static void main(String args[]) {
    if(args.length == 2) {
        try {
            SASyncExample kqmlagent = new SASyncExample(args[0],args[1]);
            kqmlagent.setPriority(Thread.NORM_PRIORITY);
            kqmlagent.start();
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
    else {
        System.out.println("AddressFilePath should be passed as arguments");
    }
}

}

//*****
class monitor implements Runnable {
    Thread t;
    int typeAgent=-1;
    int num_iteracions=15;
    SASyncFrameExample frameExemple;
    String nomFrame;

    monitor(int tipusAgent,Frame context,String nom) {
        typeAgent=tipusAgent;
        nomFrame=nom;
        frameExemple=(SASyncFrameExample)context;
        t=new Thread(this,"Monitor");
        System.out.println("Estem al fil: "+t.getName());
        t.start();
    }
}

```

```

    }

    //Codi del fil monitor
    public void run() {
        if (typeAgent == 1) {
            for(int i=num_iteracions; i>0; i--) {
                try {
                    System.out.println("Vaig a dormir");
                    Thread.sleep(15000);
                    System.out.println("Hola, ja estic despert!");
                    System.out.println("Queden "+i+" iteracions");
                    frameExemple.creaMissatgeBrossa();
                } catch (InterruptedException e) { }
            }
        }
        if (typeAgent == 2) {
            System.out.println("Soc el THREAD del Servidor");
            frameExemple.iniPosicio();
            posicions _f = new posicions(nomFrame);
            _f.start();
            int x,y;

            _f.pinta(2,frameExemple.getRX(),frameExemple.getRY());
            _f.repaint();
            while(true) {
                try {
                    t.sleep(300);
                    if (frameExemple.HiHaRubbish()) {
                        int varctrl=-1;
                        x=frameExemple.getX();
                        y=frameExemple.getY();
                        _f.pinta2(frameExemple.getRX(),frameExemple.getRY(),x,y);
                        _f.repaint();
                        do {

                            varctrl=frameExemple.recollirRubbish(x,y,_f);
                            t.sleep(50);
                        } while ((varctrl!=1) && (varctrl!=2));
                        if (varctrl==1) {
                            System.out.println("MOLT BE, BROSSA RECOLLIDA");
                            _f.pinta(3,x,y);
                            _f.repaint();
                        }
                        else if (varctrl==2) {
                            System.out.println("BROSSA RECOLLIDA... PER ALGU
ALTRE");
                        }
                    }
                } catch (InterruptedException e) { }
            }
        }
    }
}

```

9.4.2.2. Interfície gràfica de l'agent

```
//*****
// File: /mnt/d/pfc/JATLite/jatlitelinux/rubbish/exemple/SASyncFrameExample.java

package rubbish.exemple;

import java.awt.*;
import java.awt.event.*;
import java.util.Random;
import java.lang.Thread;
import java.util.*;
import java.lang.*;

import BaseLayer.*;
import Abstract.*;
import KQMLLayer.*;

public class SASyncFrameExample extends Frame implements ActionListener,ItemListener {

    AgentAction _agentAction;
    Choice _connection;
    TextArea _sendArea;
    TextArea _receiveArea;
    String _currentReceiver;
    private int typeAgent;
    Thread t;
    boolean feina=false;
    int x=1,y=1;
    int xrub,yrub;
    boolean podemRecollirBrossa=true;

    public SASyncFrameExample(String name,AgentAction awf) {
        super(name);
        _agentAction = awf;
        typeAgent=-1;

        Panel buttonPanel = new Panel();
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();

        buttonPanel.setLayout(gridbag);

        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;

        _connection = new Choice();
        _connection.addItemListener(this);
        // choice malfunction. Add two dummy choice
        _connection.add("");
        _connection.add("");

        ConnectionTable ctable = awf.getConnectionTable();
        Enumeration e = ctable.keys();
        while(e.hasMoreElements()) {
            String agentname = (String)e.nextElement();
            _connection.addItem(agentname);
        }
        // set default as the first connection
        if(!ctable.isEmpty()) {
            _connection.select(0);
            _currentReceiver = _connection.getSelectedItem();
        }

        gridbag.setConstraints(_connection,c);
        buttonPanel.add(_connection);

        c.weightx = 0.5;
        makeButton("Connect",gridbag,buttonPanel,c);
        makeButton("Compose",gridbag,buttonPanel,c);
        makeButton("Send",gridbag,buttonPanel,c);

        c.gridwidth = GridBagConstraints.REMAINDER;
        makeButton("Quit",gridbag,buttonPanel,c);
    }
}
```

```

        add("North",buttonPanel);

        Font f=new Font("Helvetica",Font.PLAIN, 10);
        setFont(f);

        _receiveArea = new TextArea("Receiving Area\n",10,40);
        _receiveArea.setEditable(false);
        add("Center",_receiveArea);
        _sendArea = new TextArea("Sending Area\n",10,40);
        add("South",_sendArea);

        setSize(269,315);
        setVisible(true);

        System.out.println("Ini acabada.");
    }

    protected void makeButton(String name,GridBagLayout gridbag,Panel p,
GridBagConstraints c) {
        Button button = new Button(name);
        button.addActionListener(this);
        gridbag.setConstraints(button,c);
        p.add(button);
    }

    public void actionPerformed(ActionEvent evt) {
        String arg = evt.getActionCommand();

        System.out.println("Event Activat: "+evt.toString());
        System.out.println("Accio: "+arg);
        if("Connect".equals(arg)) {
            SASyncConnectionDlg connectionDlg = new SASyncConnectionDlg(this,
_agentAction.getAddressTable(),_agentAction.getConnectionTable());
            connectionDlg.showDlg();
        }
        else
            if("Compose".equals(arg)) {
                String defaultMessage = "tell \r\n";
                defaultMessage = defaultMessage + " :sender " + getTitle() + "
\r\n";
                defaultMessage = defaultMessage + " :receiver " + _currentReceiver
+ " \r\n";
                defaultMessage = defaultMessage + " :language KQML \r\n";
                defaultMessage = defaultMessage + " :content (      )";
                _sendArea.setText(defaultMessage);
                System.out.println("El missatge composat es: ");
                System.out.println(defaultMessage);
            }
            else
                if("Send".equals(arg)) {
                    String sendstr = _sendArea.getText();
                    System.out.println("La part llegida per enviar es:");
                    System.out.println(sendstr);
                    int strlen = sendstr.length();
                    String cstr = "(";
                    cstr = cstr + filtering(sendstr) + ")";
                    System.out.println("i despres de filtrar:");
                    System.out.println(cstr);
                    try {
                        _agentAction.sendMessage(_currentReceiver,cstr);
                        System.out.println("Missatge per enviar:");
                        System.out.println(cstr);
                    } catch (ConnectionException c) {
                        System.out.println(c.toString());
                    }
                }
                else
                    if("Quit".equals(arg)) {
                        _agentAction.processMessage("Quit",null);
                        dispose();
                        System.exit(0);
                    }
            }
    }

```

```

public void itemStateChanged(ItemEvent evt) {
    _currentReceiver = _connection.getSelectedItem();
    notifyMessage("Receiver is set to " + _currentReceiver);
}

protected String filtering(String sendstr) {
    String retstr = "";
    int strlen = sendstr.length();
    for(int i=0; i < strlen; i++) {
        char c = sendstr.charAt(i);
        if((c != '\r') && (c != '\n')) {
            retstr = retstr + c;
        }
    }
    return retstr;
}

public void notifyError(String error) {
    _receiveArea.append(error + "\n");
}

public void notifyMessage(String msg) {
    _receiveArea.append(msg + "\n");
}

public void processMessage(String command,String args[]) {
    if(command.equals("Connect")) {
        System.out.println("Try to connect " + args[0]);
        Address addr = new Address(args[0],args[1],(Integer.valueOf(args[2])).intValue(),args[3],args[4]);
        if(addr.getPort() == -1) {
            notifyMessage("Receiver " + args[0] + " does not have ServerSocket and it is off-line. Connection aborted");
            return;
        }
        try {
            _agentAction.createReceiverThread(args[0],Thread.NORM_PRIORITY);
            notifyMessage("Receiver is set to " + args[0]);
            System.out.println("Notify finish");
        } catch (Exception e) {
            System.out.println(e.toString());
            notifyError(e.toString());
        }
    }
    else
        if(command.equals("Disconnect")) {
            ConnectionTable ctable = _agentAction.getConnectionTable();
            ctable.removeConnection(args[0]);
        }
        else
            if(command.equals("RemoveConnection")) {
                try {
                    _connection.remove(args[0]);
                } catch (Exception e) {}
            }
            else
                if(command.equals("RemoveAddress")) {
                    AddressTable atable = _agentAction.getAddressTable();
                    Address addr = atable.getAddress(args[0]);
                    if(addr != null) {
                        atable.remove(addr.getID());
                    }
                }
                else
                    if(command.equals("AddAddress")) {
                        AddressTable atable = _agentAction.getAddressTable();
                        Address addr = new Address(args[0],args[1],(Integer.valueOf(args[2])).intValue(),args[3],args[4]);
                        // if same name address is already defined, remove
                        Address tmp = atable.getAddress(args[0]);
                        if(tmp != null) {

```

```

        atable.remove(tmp.getID());
    }
    atable.addAddress(addr);
}
else
if(command.equals("AddConnection")) {
    addConnectionItem(args[0]);
}
}

protected void addConnectionItem(String conid) {
    _connection.add(conid);
    _connection.select(conid);
}

public void processMessage(Object o) {
    try {
        KQMLmessage kqml = (KQMLmessage)o;
        String msg = kqml.getReadString();
        // There seems to be a bug in the 'append' method of TextArea.
        _receiveArea.setText("\r\n-----\r\n" + msg +
"\r\n");
    } catch (Exception e) {
        _receiveArea.append(e.toString());
    }
}

public void setClient() {
    typeAgent=1;
    System.out.println("ES UN CLIENT "+getName());
}

public void setServidor() {
    typeAgent=2;
    System.out.println("ES UN SERVIDOR "+getName());
}

public void iniPosicio() {
    Random rand = new Random( (long)System.currentTimeMillis() );
    int f = Math.abs(rand.nextInt() % 190);
    rand = new Random( (long)System.currentTimeMillis() );
    int c = Math.abs(rand.nextInt() % 190);
    moure(f,c);
}

public void creaMissatgeBrossa() {
    Random rand = new Random( (long)System.currentTimeMillis() );
    int f = Math.abs(rand.nextInt() % 190);
    int c = Math.abs(rand.nextInt() % 190);

    String Message = "brossa \r\n";
    Message = Message + " :sender " + getTitle() + " \r\n";
    Message = Message + " :receiver All \r\n";
    Message = Message + " :language KQML \r\n";
    Message = Message + " :content "+f+","+c;

    try {
        _sendArea.setText(Message);

        String sendstr=_sendArea.getText();
        String msg="(";
        msg=msg+filtering(sendstr)+")";

        _agentAction.broadCast(msg);
        System.out.println("El missatge filtrat es: ");
        System.out.println(msg);

    } catch (ConnectionException ce) {
        _sendArea.append(ce.toString());
    }
}
}

```

```

public void construirMissatgeBrossaRecollida(int xr, int yr) {

    String Message = "recollida \r\n";
    Message = Message + " :sender " + getTitle() + " \r\n";
    Message = Message + " :receiver All \r\n";
    Message = Message + " :language KQML \r\n";
    Message = Message + " :content "+xr+","+yr;

    try {
        _sendArea.setText(Message);

        String sendstr=_sendArea.getText();
        String msg="(";
        msg=msg+filtering(sendstr)+")";

        _agentAction.broadCast(msg);
        System.out.println("El missatge filtrat es: ");
        System.out.println(msg);

    } catch (ConnectionException ce) {
        _sendArea.append(ce.toString());
    }
}

public void setRubbish(String position) {
    System.out.println("OK! JA TENIM FEINA");
    System.out.println("Posicio actual: "+x+","+y);

    podemRecollirBrossa=true;
    int strlen = position.length();
    String numstr="";
    int num[]=new int[4];
    int j=0;
    for(int i=0; i < strlen; i++) {
        char c = position.charAt(i);
        if (c==',') {
            num[j]=Integer.parseInt(numstr);
            numstr="";
            j++;
        }
        if((c != '\r') && (c != '\n') && (c != ',') && (c != ' ')) {
            numstr = numstr + c;
        }
    }
    num[j]=Integer.parseInt(numstr);

    xrub=num[0];
    yrub=num[1];
    feina=true;
    System.out.println("Posicio brossa (integers): "+xrub+" "+yrub);
}

public int getX() {return xrub;}
public int getY() {return yrub;}
public int getRX() {return x;}
public int getRY() {return y;}
public boolean HiHaRubbish() {
    boolean valor=feina;
    if (feina)
        feina=false;
    return valor;
}

public void moure(int a, int b) {
    x=a;
    y=b;
}

```

```

public void aturarRecollida() {
    podemRecollirBrossa=false;
}

public int recollirRubbish(int x1, int y1, posicions _p) {
    System.out.println("Anem a recollir la brossa de "+x1+", "+y1);
    System.out.println("i som a "+x+", "+y);
    int dx=x1-x;
    int dy=y1-y;
    int absdx=Math.abs(dx);
    int absdy=Math.abs(dy);

    if (podemRecollirBrossa) {
        if ((dx==0) && (dy==0))
        {
            construirMissatgeBrossaRecollida(x,y);
            return 1;
        }
        else if (dx==0)
            moure(x,y+(dy/absdy));
        else if (dy==0)
            moure(x+(dx/absdx),y);
        else if (absdx<absdy)
            moure(x+(dx/absdx),y);
        else if (absdx>=absdy)
            moure(x,y+(dy/absdy));
    }
    else
    {
        return 2;
    }

    System.out.println("MOGUTS A "+x+", "+y);
    _p.pinta2(getX(),getY(),x,y);
    _p.repaint();
    System.out.println("Intentem cridar el metode pinta desde recollirRubbish");

    return 0;
}
}

```

9.4.2.3. Interficie gràfica de la zona de treball

```
//*****
// File: /mnt/d/pfc/JATLite/jatlitelinux/rubbish/exemple/posicions.java

package rubbish.exemple;

import java.*;
import java.awt.*;

class posicions extends Frame {

    int posx=100;
    int posy=100;
    int posx1;
    int posy1;
    int tipusFigura=1;
    int tipusFigura1=-1;

    posicions(String title) {
        setSize(207,220);
        setTitle(title);
        setVisible(true);
        setBackground(Color.white);
        setForeground(Color.blue);
        Font f=new Font ("Helvetica",Font.PLAIN,10);
        setFont(f);
    }

    /*
    Thread _ta;

    public void run() {
        while(true) {
            repaint();
            try {
                _ta.sleep(100);
            } catch (InterruptedException ee) { }
        }
    }
    */

    public void start() {
        //_ta=new Thread("fil_pintor");
        //System.out.println("FIL PINTOR CREAT");
    }

    public void pinta(int tipus, int x1, int y1) {
        tipusFigura=tipus;
        posx=x1;
        posy=y1;
        repaint();
    }

    public void pinta2(int x1, int y1, int x2, int y2) {
        tipusFigura1=1;
        posx=x1;
        posy=y1;
        posx1=x2;
        posy1=y2;
        repaint();
    }

    public void paint(Graphics g) {
        posx=posx+5;
        posy=posy+20;

        if (tipusFigura1==1) {
            tipusFigura1=-1;
            g.setColor(Color.black);
            g.fillRect(posx1+5, posy1+20, 5, 5);
            g.setColor(Color.magenta);
            g.drawRect(posx, posy, 5, 5);
        }
        else if (tipusFigura==1) {
            g.setColor(Color.magenta);
        }
    }
}
```

```

        g.drawRect(posx, posy, 5, 5);
    }
    else if (tipusFigura == 2)
    {
        g.setColor(Color.black);
        g.fillRect(posx, posy, 5, 5);
    }
    else if (tipusFigura == 3)
    {
        g.setColor(Color.blue);
        g.fillRect(posx, posy, 5, 5);
    }

    //g.drawRect(posx+5, posy+20, 4, 4);

    //for (int i=0; i<4; i++) {
    //    g.drawRect(i+5, i+20, 190, 190);
    //}
}
}

```

9.5. Possibles aplicacions de JATLite

Les funcions basades en Internet (email i FTP) que proporciona JATLite, més les classes i mètodes que proporciona Java (http, applets, accés a URLs, ...) fan que es tingui un ampli ventall de possibles implementacions. Es pot pensar en assistents personals, agendes, assistents de navegació, agents de recuperació d'informació, teledocumentació, etc.

Més enllà de les aplicacions d'Internet, es poden realitzar aplicacions en altres àmbits. Podríem pensar en aplicacions de gestió de serveis. Aquestes aplicacions són, però, difícils d'implementar perquè JATLite només proporciona les eines bàsiques de comunicació, i se l'usuari necessita aplicar estratègies de cooperació/negociació per tal d'aconseguir algun objectiu, les ha d'implementar en la seva totalitat.

Com es veurà més endavant, altres eines de desenvolupament proporcionen llibries d'utilitats o estratègies que hom pot importar i adequar-les a les necessitats de l'aplicació.

La flexibilitat és la característica que diferencia aquest software i, depenent del coneixement de Java que tingui el programador, serà més o menys senzilles d'implementar.

Una altra possible aplicació és l'agentificació (v.Figura 9.7) d'altres programes realitzats prèviament.

Com no s'implementa cap classe abstracta de tipus *agent*, la flexibilitat permet d'implementar aquesta tècnica a banda de la pròpia implementació d'un agent JATLite. Aquesta implementació ha de tenir en compte que en última instància, la transmissió de dades entre agents es realitza en caràcters ASCII. Els desenvolupadors del software JATLite senyalen que s'ha realitzat agentificació de software heretat en C, C++, i Lisp afegint un embolcall als programes.

JATLite ha estat usat per a la creació d'agents d'interfície per una àmplia varietat d'aplicacions de software comercial CAD, prototipus d'agents d'anàlisi financera (Yamaichi Information Systems Inc.), agents de cerca d'informació (Tektronix). Ha estat usat també en aplicacions de recerca aplicades a l'organització d'empreses, amb cooperació de Toshiba.

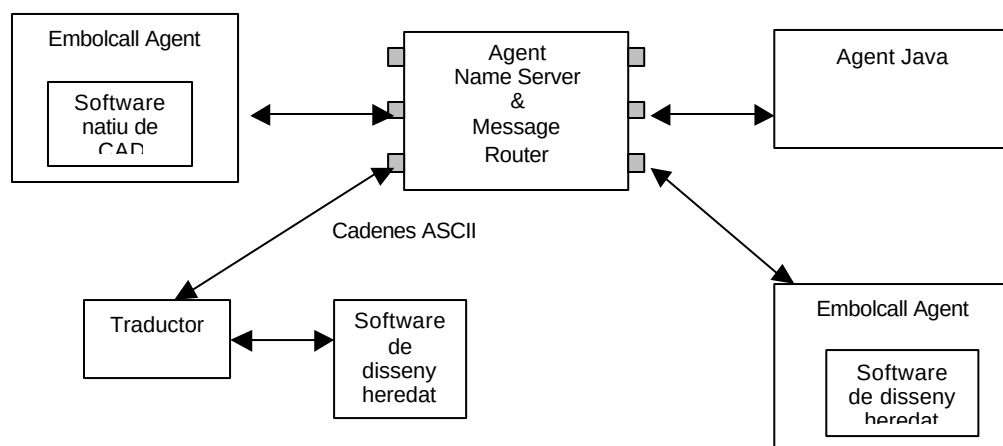


Figura 9.7. Aplicacions de JATLite: herència de codi, traducció i reescriptura.

Java-based Agent Framework for Multiagent Systems (JAFMAS)

Alan Galan, University of Cincinnati
(Department of Electrical and Computer Engineering and Computer Science)

<http://www.ececs.uc.edu/~abaker/JAFMAS/>

David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

10. ESTUDI DE JAFMAS	116
10.1. DESCRIPCIÓ GENERAL	118
10.2. METODOLOGIA DE DESENVOLUPAMENT	119
10.2.1. Comunicació	119
10.2.2. Interacció	121
10.2.3. Coherència i Coordinació	122
10.2.4. Passos per desenvolupar una aplicació JAFMAS	124
10.3. ARQUITECTURA I IMPLEMENTACIÓ EN JAVA	126
10.3.1. Nivell de protocol de comunicació	128
10.3.1.1. Broadcast	128
10.3.1.2. Comunicació directa	128
10.3.2. Nivell lingüístic	129
10.3.2.1. Missatges	129
10.3.2.2. Cues de missatges	129
10.3.3. Nivell Social	130
10.3.3.1. Requerir recursos	130
10.3.3.2. Converses	131
10.3.3.3. Regles de conversa	131
10.3.3.4. Encaminament de missatges	133
10.3.4. Nivell multiagent	134
10.3.5. Interfícies d'Agents	135
10.3.5.1. Interfície de creació d'agents	135
10.3.5.2. Interfície de l'agent	135
10.3.5.3. Interfície de converses	137
10.4. DESENVOLUPAMENT D'UNA APLICACIÓ	138
10.4.1. Descripció de l'aplicació	138
10.4.1.1. Connexions	138
10.4.1.2. Creació dels agents	139
10.4.1.3. Funcionament dels agents	140
10.4.1.4. Comunicació	142
10.4.1.5. Resultats obtinguts	143
10.4.2. Codi complet	144
10.4.2.1. Creació dels agents	144
10.4.2.2. Agent Boss	145
10.4.2.3. Agent Robot	147
10.4.2.4. Tauler del robot	151
10.5. APLICACIONS PRÀCTIQUES	153
10.6. CONCLUSIONS	154

10.1. Descripció general

JAFMAS (*Java-based Agent Framework for Multi-Agent Systems*) és una estructura que intenta representar i desenvolupar el coneixement en la cooperació i en els protocols de sistemes multiagent. Aquesta estructura permet als agents treballar conjuntament i, de forma coherent, aconseguir els seus objectius i els de la comunitat multiagent com un tot.

Més endavant veurem alguns dels problemes en la coordinació dels sistemes multiagent i com es poden solucionar. Es discuteixen també algunes teories sobre com haurien de ser els principis per a implementar els agents per tal de desenvolupar sistemes multiagent. Aquests principis asseguren *comunicació*, *coordinació* i *cooperació* entre agents i ajuden als agents a aconseguir una solució per al problema.

La majoria d'eines de desenvolupament de SMA pateixen l'inconvenient d'implementar un *Agent Name Service* (ANS) central pel qual passen tots els missatges o, té la informació/localització de tots els agents de la comunitat.

Els agents JAFMAS mantenen una agenda de tots els interlocutors als quals es pot accedir, d'aquesta manera es pot realitzar una comunicació més eficient. Aquesta és una decisió de disseny amb algunes avantatges (tolerància a fallides, més rapidesa de transmissió, escalabilitat, flexibilitat, etc.) però també té algunes desavantatges, doncs l'agent ha de mantenir una estructura i assegurar que tota la seva informació és correcta quan s'ha d'utilitzar.

Característiques de JAFMAS	
Llenguatge de programació	Java (JDK 1.1)
Cada agent treballa per sí sol?	Sí (cadascun és un thread)
Comunicació entre agents	Directe o Multicast
Mitjans de comunicació directa?	Utilitza RMI de Java i MulticastSocket
ANS	Mantingut per cada agent.
És escalable?	Sí
ACL	KQML
Proporciona conceptes de coordinació?	Sí, converses.
Característiques de seguretat	Lligat a la seguretat de Java.

Els agents poden fer ús de dos tipus de comunicació: *directe* o *multicast*.

La comunicació multicast permet als agents JAFMAS identificar altres agents en el sistema. Cada vegada que aquests agents JAFMAS necessiten algun servei, han d'emetre la petició que desitgen o requeriment, a la comunitat d'agents. Cada agent que pot proporcionar el servei demanat, retorna la resposta.

La característica principal de la comunicació *multicast* és que ajuda a implementar sistemes escalables, tals com quan un agent pren part d'un sistema, comença a treballar amb altres agents o bé deixa el sistema un cop ha acabat la tasca que li estava encarregada, sense afectar l'operació del sistema.

Tant JATLite com JAFMAS implementen un ACL basat en *speech-act*, en el qual cadascun dels missatges que s'intercanvien els agents tenen un format i una semàntica basats en la definició de KQML.

10.2. Metodologia de desenvolupament

JAFMAS defineix una metodologia genèrica per al desenvolupament de les aplicacions multiagents i proporciona un conjunt de serveis que alliberen a l'investigador de l'esforç de programar mecanismes de *coordinació* des del principi.

Els mecanismes de JAFMAS i la seva implementació es poden fer correspondre en nivells d'abstracció. Per JAFMAS es tenen tres nivells lògics: en el nivell més alt es té la *coordinació/coherència* entre els agents, en un nivell més baix es tenen les regles que fan possible la coordinació/coherència, però aquest necessita d'un nivell més baix que proporciona les facilitats de *comunicació* (v. Figura 10.1).

JAFMAS proporciona un comportament intel·ligent de coordinació entre una col·lecció d'agents intel·ligents que formen el sistema multiagent, per tal de coordinar llurs coneixements, plans, objectius conjunts, etc. Així poden dur a terme accions els resultats de les quals, generen una solució coherent al problema plantejat (*coherència*).

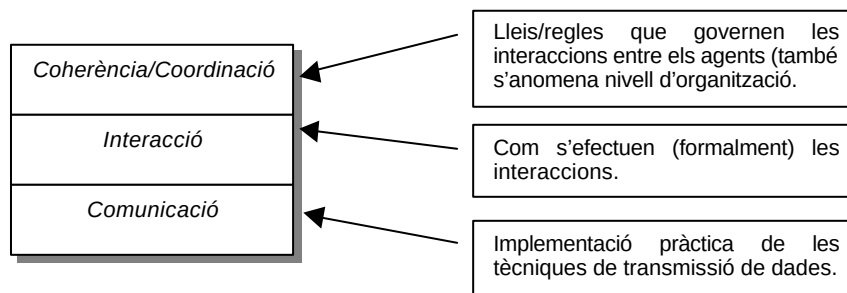


Figura 10.1: Nivell lògics que defineix JAFMAS com a base de desenvolupament de la seva metodologia de treball.

10.2.1. Comunicació

La comunicació permet als agents intercanviar informació dins del sistema multi-agent, en la base que tots coordinen les seves accions i cooperen els uns amb els altres. El mecanisme de comunicació usat sovint depèn del sistema i de les circumstàncies en les quals els agents intercanvien la informació. En algunes situacions, les sortides d'alguns agents són les entrades dels altres, o bé l'agent no coneix la identitat ni adreces dels altres agents del sistema. En aquests casos, la comunicació per difusió (*broadcast*) és més útil que la comunicació directa. Quan cal donar una difusió a un o més missatges (p.e. per enviar un missatge d'error a tots els agents, etc.), el *broadcast* és el tipus de comunicació idònia. Però si hi ha un nombre gran d'agents al sistema, la comunicació *broadcast* pot afegir una càrrega al sistema que cal avaluar en cada cas depenent de la quantitat d'agents. En altres situacions on un agent té un diàleg amb un agent particular, i sap exactament a qui envia el missatge, la comunicació directa té més sentit.

Per tal de proporcionar l'aplicació entre les dues formes de comunicació, JAFMAS suporta ambdós, és a dir, tan la comunicació *directa* com la *broadcast*. També hauria de

permetre a l'usuari combinar aquestes dues aproximacions per a formar *sistemes federats*.

Les eines d'implementació de JAFMAS poden proporcionar la comunicació bàsica necessària en un sistema multiagent, si es combina tant el mode de comunicació broadcast com la directa. Ja s'ha dit que aquests mecanismes de comunicació depenen del sistema i de les circumstàncies en les quals es troben els agents. Quan l'output d'un agent és l'input de molts altres agents, o l'agent no coneix quins agents hi ha al sistema, JAFMAS permet l'ús de links multicast per a fer una comunicació en broadcast. Per altra banda, quan és segur que el missatge output d'un agent està sent enviat a alguns agents coneguts, la comunicació consisteix en links directes.

Com JAFMAS utilitza RMI de Java per al mecanisme de comunicació directa, els agents poden transporta objectes o fins i tot, altres agents per la xarxa.

La comunicació broadcast facilita la comunicació directa. Els agents en l'estructura poden estar limitats al coneixement de la comunitat on existeixen. Així, els missatges broadcast ajuden a establir la identitat i capacitats d'altres agents en la comunitat. Els mètodes *specification-sharing* i *contract-net* suporten els missatges broadcast, per tal de demanar serveis específics o d'advertir de la seva presència a la comunitat. Basat en aquesta informació, primer decideixen a qui poden permetre una conversa, poden permetre una comunicació directa si saben amb qui s'estableix el link físic. Més enllà d'això, els missatges broadcast poden se enviats per a demanar certes accions a d'altres agents de la comunitat.

El mode de comunicació broadcast també permet la implementació del *subject-based addressing*. Cada agent es pot subscriure a un grup (o subjecte) i cada missatge proposat pel subjecte és enviat només als agents subscrits a aquest grup particular (o subjecte). Els agents es poden unir (subscriure) dinàmicament o deixar un grup. Tan aviat com els agents s'uneixen al grup, esdevenen destinataris de tots els missatges proposats pel grup i quan es deixa el grup, automàticament deixen de rebre aquests missatges.

10.2.2. Interacció

Segons Gasser, la interacció és un tipus d'accions col·lectives en les quals un agent duu a terme una acció o pren una decisió que ha estat influenciada per la presència o coneixement d'un altre agent. La natura distribuïda i heterogènia del sistema multi-agent fa de la implementació del mecanisme d'interacció entre agents un procés difícil. En la comunitat de sistemes multiagent el llenguatge basat en la parla (*speech-act*) és el mètode més comú per a formalitzar les accions lingüístiques.

Implementar el tipus de missatge del *speech-act* en JAFMAS permet al llenguatge de comunicació transmetre models d'alt nivell d'interaccions inter-agents. Cada missatge pot ser associat a una *performative* la qual és el component de l'acte de parla (*speech-act*). La *performative* determina què es pot fer o realitzar amb el contingut d'un missatge. Els agents poden llavors transportar creences, coneixement o intencions. Els missatges poden dur informació sobre l'agent emissor, sobre el seu coneixement, sobre els seus objectius, les seves intencions, desitjos, per tal d'arribar a un objectiu mutu amb el receptor.

Els missatges poden ser de dos tipus: missatges de *declaració*, que permeten a l'agent declarar la seva presència i les seves possibilitats i, missatges de *contingut*, els quals són usats per a enviar coneixement.

10.2.3. Coherència i Coordinació

Gasser diferencia entre coherència i coordinació referint-se a coherència quan discuteix com de bé es comporta el sistema sencer mentre resol un problema i examina el comportament del sistema com a unitat. La coordinació és la propietat d'interacció entre un conjunt d'agents treballant sobre una acció col·lectiva.

Sense coordinació els beneficis d'un problema descentralitzat desapareixen i el treball de la comunitat pot esdevenir caòtic.

Ambdós termes estan relacionats en el sentit que una millor coordinació dona com a resultat una solució millor i més coherent al problema. Per tal de donar una solució coherent, els objectius del procés de coordinació són assegurar que totes les parts del problema global són incloses en les activitats d'un agent (almenys un), que els agents interactuen de manera que es permeti el desenvolupament i integració de les activitats en una solució global, que els membres de l'equip actuïn en el propòsit de forma consistent, i finalment, que tot això es pugui aconseguir amb les fonts disponibles.

Segons Jennings, les raons per les quals els agents necessiten estar coordinats són les següents:

- a) Hi ha dependència entre les accions dels agents, la qual cosa succeeix quan els objectius individuals dels agents estan relacionats.
- b) Hi ha necessitat de trobar limitacions globals. Aquestes existeixen quan la solució que està sent desenvolupada pel sistema ha de satisfer certes condicions per a estimar-ne l'èxit.
- c) Els agents individuals no tenen suficient competència, recursos o informació per a resoldre un problema sencer. Alguns problemes no es poden solucionar amb agents que treballen aïlladament perquè no tenen l'experiència necessària ni la informació adequada per a fer-ho (completa).

Fins i tot, quan els agents individuals poden treballar independentment, la coordinació no és essencial, ja que la informació descoberta per cada agent pot ser d'ús suficient per a un altre agent, de manera que dos agents poden resoldre el problema més ràpid.

Si tots els agents del sistema poguessin tenir un coneixement complet dels objectius, accions i interaccions dels membres de la comunitat, i tinguessin un poder de processat infinit, seria possible conèixer exactament què estava fent un agent al present, i què està intentant de fer en un futur. En aquests casos, seria possible evitar conflictes i esforços redundants i el sistema estaria perfectament coordinat.

Malgrat tot, el coneixement complet no és factible perquè les limitacions dels sistemes de computació fan impossible que els agents puguin estar informats constantment i a l'instant de tots les evolucions.

Corkill i Lesser creuen que el problema de coordinació pot ser minimitzat tenint coneixement sobre els processos d'interacció que es donen entre els agents. Aquest coneixement es basa en la competència per a resoldre un problema d'un sistema multiagent en oposició a la resolució del mateix problema per part d'un agent individual. En aquesta aproximació a la coordinació, els agents creen un pla que especifica les seves accions futures i interaccions respecte de l'assoliment d'un objectiu particular. El poder de la Teoria de la parla (speech-act) ha estat explotat per a formalitzar els plans dels agents a través de *converses*.

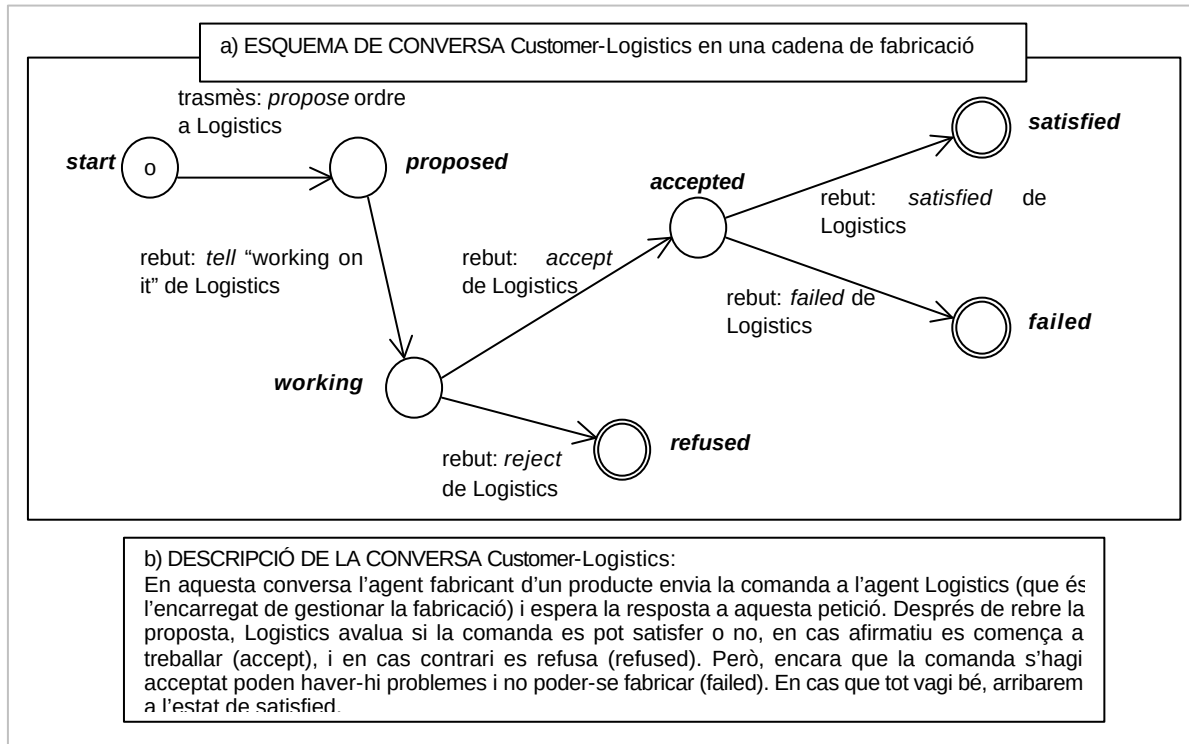


Figura 10.2: a) Veiem l'organització en forma de graf d'una conversa; b) la descripció del funcionament il·lustra el significat dels estats i llurs transicions

Una conversa és el pla d'un agent per aconseguir algun objectiu, basat en interaccions amb altres agents (v. Figura 10.2). Les converses d'agents són visualitzades en termes d'autòmats. Prenen en consideració plantejaments alternatius a seguir durant l'execució d'accions/reaccions d'altres agents. El llenguatge de converses és basat en la noció explícita d'estat. Les accions alternatives produeixen diferents estats i l'estat d'una conversa caracteritzarà les actuacions/reaccions en el moment següent. L'execució de la conversa comença en un estat inicial i acaba en un estat final. Durant la conversa, els agents poden intercanviar missatges d'acord a convenis mutus, poden canviar d'estats o bé realitzar accions locals.

Com cada conversa JAFMAS està dissenyada per a tenir el seu propi fil d'execució, els agents poden tenir múltiples converses al mateix temps. La seva funció és implementar la cooperació i coordinació entre agents.

Els autòmats proporcionen consistència i coherència de les converses d'agents i poden verificar-se amb models matemàtics. La metodologia JAFMAS proposa també l'anàlisi de les converses d'agents per a verificar la consistència lògica i coherència dels SMA. Suggereix una anàlisi matemàtica dels autòmats representant converses d'agents per assegurar que les interaccions entre ells es fan de forma correcta.

10.2.4. Passos per desenvolupar una aplicació JAFMAS

En la construcció dels sistemes multi-agent, ha sorgit una metodologia, que està formada per **cinc** passos:

- I) **Identificar els agents.** Els usuaris tenen una llista d'intencions a satisfer pel sistema. Es comença per considerar les entitats varies que interaccionaran les unies amb les altres per aconseguir l'objectiu comú. Aquestes entitats representen als agents del sistema. Un cop identificats, s'intenta clarificar els objectius de cada agent i els serveis que proporcionaran. Es divideixen els agents en diferents categories, agrupant a tots aquells amb objectius i serveis similars en una mateixa categoria. Les categories ajuden a l'usuari a identificar les classes d'agents en l'aplicació concreta.

- II) **Identificar les converses.** S'usa la Teoria d'Organització (*Organization Theory*) per a identificar les converses dels agents i estableix que el problema de coordinació pot ser abordat amb el coneixement dels processos d'interacció entre agents. Aquest coneixement es basa en la capacitat de resoldre un problema del sistema sencer i no dels agents específics. El món està format per objectes intel·ligents i es comença per modelar la interacció entre aquests objectes en forma de converses. Una conversa és un pla d'un agent per assolir un objectiu, basat en interaccions amb altres agents. Com a plans, prenen en consideració tractaments alternatius d'accions durant l'execució, basant-se en accions/reaccions alternatives d'altres agents. Admeten autòmats. Identifiquen cada possible conversa, que pot ser permesa a un agent i la representen desenvolupant un autòmat per a cadascuna. Els autòmats identifiquen les diferents classes de conversa de l'aplicació.

- III) **Identificar les regles de conversa.** Cada conversa és representada per un autòmat (v. apartat 10.2.3). Les accions alternatives d'un agent produeixen diferents estats en la conversa, i l'estat actual de la conversa influeix en la forma en què actuarà/reaccionarà l'agent en el moment següent. Les converses són regles basades en descripcions d'allò que un agent fa en determinades situacions. Es necessiten identificar aquestes accions de l'agent i les condicions que produeixen l'execució de la regla. Tot això ens ajuda a identificar les regles de conversa per a cada conversa.

- IV) **Analitzar el model de conversa.** Per tal de dissenyar un sistema multi-agent que trobi una solució coherent al problema del sistema, és important fer una anàlisi de la consistència lògica de totes les converses dels agents. Haurien de ser analitzats per a verificar la coherència del sistema. Alguns exemples són *Petri Nets* o *Dooley graphs*, com a eines útils per a analitzar el sistema i les converses. Si hi ha alguna inconsistència al model, el disseny torna al pas II) de la metodologia per a redefinir les converses.
- V) **Implementació SMA.** Com a pas final, es necessita escollir una eina adequada per a la implementació de l'aplicació multi-agent, que assegurí la comunicació, inter-operació i la coordinació com a suports. Les eines haurien de proporcionar suport per a la comunicació directa i multicast, la creació d'agents propis, missatges basats en la parla (*speech-act*), la iniciació de la conversa en el seu propi fil i desenvolupament de les regles de conversa. JAFMAS proporciona un conjunt de classes de Java que poden ser usats per a codificar les aplicacions SMA, com a part d'aquesta metodologia.

10.3. Arquitectura i Implementació en Java

Seguint la metodologia abans explicada s'assegura la comunicació, coordinació i cooperació entre els agents en un sistema multiagent dinàmic, així com l'arribada a una solució coherent. En aquest apartat es veurà com JAFMAS implementa tots els principis que s'han anat explicat en les seccions precedents.

JAFMAS defineix una jerarquia modular per tal d'implementar un sistema d'agents (v. Figura 10.3). Com ja s'ha vist, en la descripció general de la metodologia de desenvolupament es poden distingir tres nivells: comunicació, interacció i organització. Bé doncs, la base de l'arquitectura JAFMAS està composta d'aquests termes i els complementa amb elements de Java i del sistema multiagent.

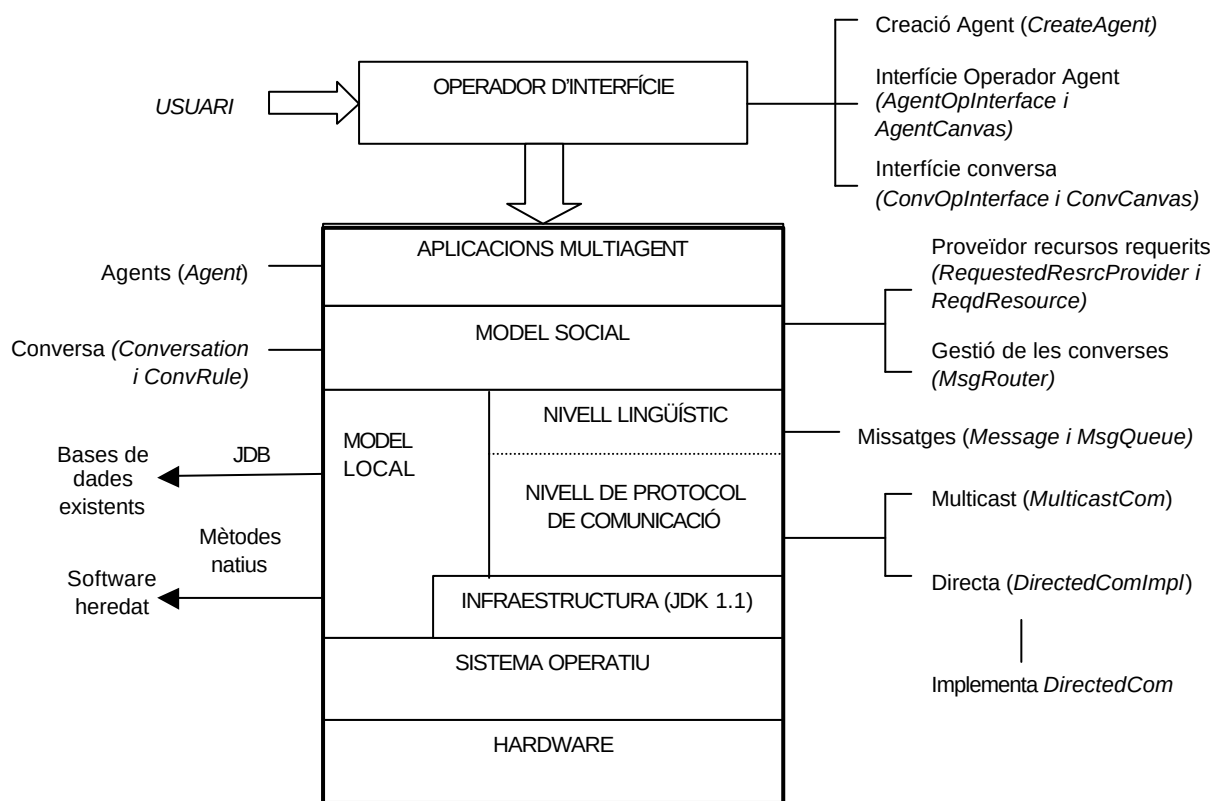


Figura 10.3: Descripció de tots els nivells de l'arquitectura JAFMAS i llurs classes implementades¹⁰

¹⁰ Extret de la documentació de JAFMAS, Capítol 4, pàg.67.

JAFMAS és desenvolupat en JDK 1.1. Proporciona setze classes de Java dividint els serveis proporcionats per aquestes classes en diferents nivells.

Els dos nivells més baixos són els d'*hardware* i *sistema operatiu* i, per sobre d'ells resideix la resta del sistema.

El *model local* és l'aplicació específica i es deixa que l'usuari la implementi com li sigui necessari. L'ús de Java com a infraestructura permet a l'usuari integrar amb aplicacions d'herència i bases de dades, mentre es desenvolupen els models locals.

Per sobre d'aquesta infraestructura queda el *nivell de comunicació*, que actualment consisteix en dos subnivells: el *nivell lingüístic* (llenguatge comú per a la comunicació d'agents independents) i *nivell de protocol de comunicació* (modes de comunicació empleats). Quan la comunicació entre agents és assegurada, els agents han d'haver pres mesures per a interactuar amb els altres de manera coordinada i cooperativa.

El *model social* de l'agent es construeix a partir del nivell de comunicació. Un cop s'han proporcionat els serveis de comunicació, lingüística i suport de coordinació, tenim tots els components necessaris i disponibles per a construir agents.

Per sobre d'aquests nivells hi ha l'*aplicació multi-agent*, que està formada per agents JAFMAS.

Els *operadors d'interfície*, proporcionats pel framework, ajuden a l'usuari a interactuar amb l'aplicació.

En les subseccions següents descriurem per separat cadascun dels elements, donant les característiques principals i la seva implementació.

10.3.1. Nivell de protocol de comunicació

JAFMAS suporta dues formes d'efectuar l'intercanvi de missatges entre els agents: *broadcast* i *comunicació directa*.

10.3.1.1. Broadcast.

Aquest tipus de comunicació s'implementa en les classes *MulticastCom* i *MulticastGroup*.

JAFMAS permet a un agent subscriure's a un o més d'un grup d'agents, i enviar-hi missatges o escoltar. A cada grup que un agent es connecta es crea un *MulticastCom*.

Cada *MulticastCom* és un fil d'execució o *thread* que proporciona Java.

Aquest tipus de comunicació implementa el protocol de comunicació UDP (part de la família TCP/IP) i utilitza la classe *MulticastSocket* de Java.

Broadcast permet a un agent no tenir un directori d'agents (*agent directory*), que mantingui totes les connexions/adreces del sistema. A més permet de créixer al sistema fàcilment. L'únic problema és la quantitat de missatges transmesos.

10.3.1.2. Comunicació directa

En la comunicació directa un agent es comunica amb un altre, on l'emissor indica explícitament el destinatari del missatge. Aquest tipus de comunicació s'implementa amb les classes *DirectedComImpl* i *DirectedCom*.

La transmissió directa utilitza una nova eina proporcionada per Java: l'RMI (*Remote Method Invocation*). Aquesta implementació permet a un client/servidor transmetre objectes sense necessitat de convertir-los en cadenes de bytes. Així, és possible de transmetre dades complexes i objectes formant part del contingut dels missatges.

L'únic problema està en l'execució pràctica del paquet RMI. S'han detectat falles d'inicialització en sistemes UNIX i Windows.

10.3.2. Nivell lingüístic

El nivell lingüístic té com a funció principal la de manegar els missatges que es transmeten els agents. Els missatges es basen en la teoria de la parla (*speech-act*) i implementa el llenguatge de comunicació KQML. Cada acte permès s'anomena *performative*.

Dins d'aquest nivell es diferencien dos aspectes: els missatges i el manegament d'aquests. En els apartats següents es veuen tots dos en detall.

10.3.2.1. Missatges

Els missatges s'implementen amb la classe *Message*. Cada instància d'aquesta classe és una *performative*.

L'usuari és lliure d'utilitzar els tipus de *performatives* que es defineixen en KQML (v. Llenguatges de comunicació), o d'utilitzar-ne de noves creades per al domini concret d'aplicació.

La classe *Message* proporciona mètodes tant per assignar com per consultar tots els paràmetres dels missatges.

JAFMAS dona facilitats per a evitar que missatges vaguin per la xarxa i cada missatge té assignats quan va ser enviat i quan es va rebre. Aquestes etiquetes però, no formen part del missatge KQML sinó que són afegits pels mètodes de transmissió dels missatges; és a dir, s'eviten discordances de missatges KQML.

Semànticament podem trobar dos tipus de missatges:

- a) Declaratius, els quals anuncien la presència d'un agent o adverteixen de les seves capacitats a un altre agent (p.e. els missatges continguts als mètodes *advertisePresence()* o *subscribeTo()*).
- b) Missatges de contingut, els quals contenen parts de la base de coneixement (p.e. el contingut de *sendMsg()*)

El contingut dels missatges pot ser una cadena de caràcters, un objecte o un programa, el qual s'encapsula dins d'una classe *Object*, i és transmet usant la capacitat del paquet Java i RMI.

10.3.2.2. Cues de missatges

JAFMAS proporciona la classe *MsgQueue* per a afegir i esborrar missatges guardats a la cua.

A més de proporcionar aquests serveis bàsics també es proporcionen altres mecanismes de cerca, per exemple, buscar missatges que hagi enviat un agent en particular, o buscar un tipus de *performative* concreta. Aquests mètodes poden combinar-se entre ells si s'escau.

10.3.3. Nivell Social

El nivell social considera aspectes de la comunicació d'un agent amb la resta per tal d'aconseguir un objectiu concret. Es consideren aspectes com, per exemple, si es pot trobar una solució a un problema i, si es troba, assegurar que aquesta és coherent amb les especificacions del domini.

Aquest nivell és el més important de tota l'arquitectura.

El comportament social d'un agent en JAFMAS és proporcionat per les classes: *RequestedResrcProvides*, *ReqdResource*, *Conversation*, *ConvRule* i *MsgRouter*.

10.3.3.1. Requerir recursos

És una utilitat molt bona.

JAFMAS proporciona unes facilitats per tal que un agent pugui demanar ajuda a altres agents per tal d'aconseguir algun objectiu.

Cada agent té una llista amb uns recursos i qui els proporciona. En qualsevol moment un agent pot requerir d'algun tipus de servei. Llavors consulta la seva llista de recursos per veure si es troba o no, i en cas afirmatiu es demana el recurs a l'agent directament. Si el recurs que l'agent necessita no es troba en la seva llista, cal trobar algun agent que ho pugui realitzar, i per això cridarem un mètode que inicia el procés de cerca (v. Figura 10.4).

Aquestes facilitats s'implementen a les classes *RequestedResrcProvider* i *ReqdResource*.

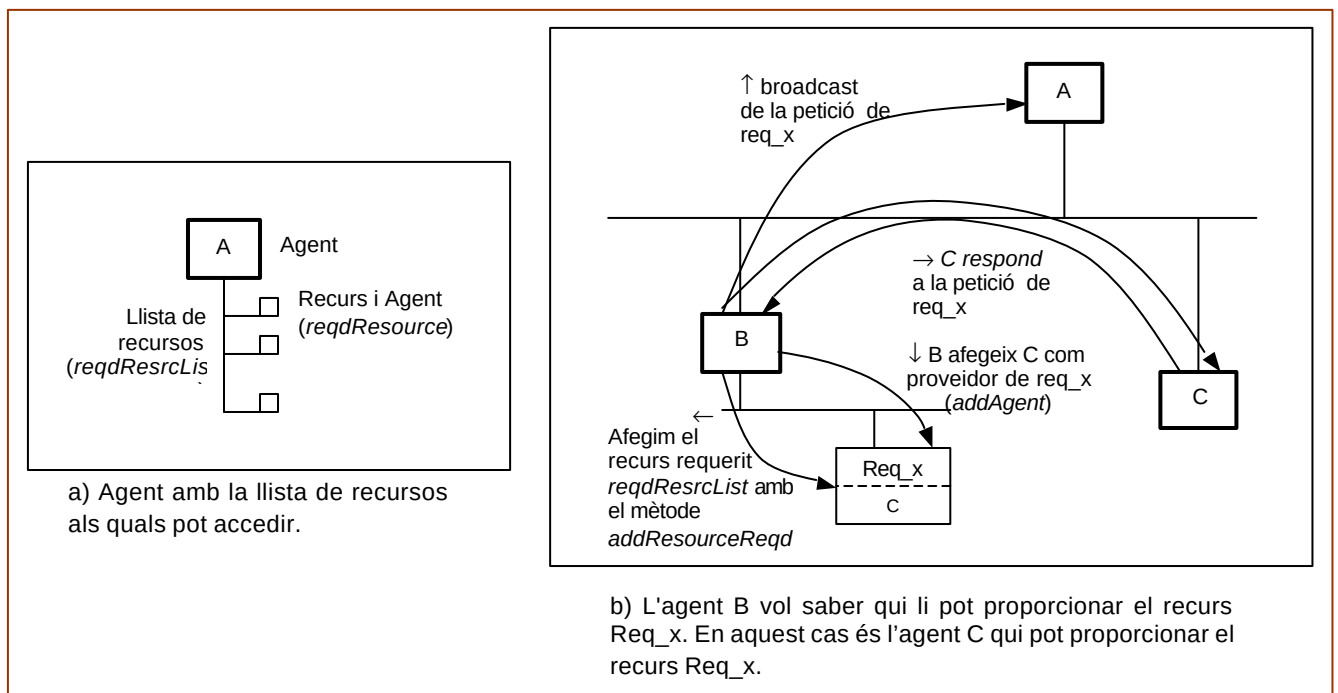


Figura 10.4: Funcionament del sistema de recursos que proporciona JAFMAS.

10.3.3.2. Converses

Una conversa és el pla d'un agent per tal d'aconseguir algun objectiu, basat en interaccions amb altres agents (assumpció cooperativa). Com que aquests plans prenen en consideració plantejaments d'accions durant l'execució, basats en accions/reaccions alternatives d'altres agents, el llenguatge que descriu les converses es basa en una noció explícita d'estat. Les accions alternatives produeixen diferents estats i l'estat actual d'una conversa influeix en la forma com l'agent actuarà/reaccionarà en un moment pròxim. El pla d'execució és sempre començat en el seu estat inicial i els estats finals descriuen situacions en les quals el pla serà acabat. Dins d'una conversa, els agents intercanvien missatges d'acord a convencions mútues, canvien d'estat i realitzen accions locals. Les converses admeten la representació d'un autòmat finit.

La classe conversa (*Conversation*) de JAFMAS extèn la classe *thread* de Java, ja que cada conversa d'un agent té el seu propi fil (*thread*) d'execució. La classe de conversa és una regla abstracta basada en la descripció d'allò que un agent fa en una determinada situació (p.ex. quan rep un missatge amb una estructura donada). Defineix interaccions en el sistema multiagent. L'usuari necessita incloure aquesta classe per tal d'implementar classes de conversa d'aplicació específica, és a dir, cal implementar la classe i els mètodes genèrics que inclou (v. exemple de la secció següent).

Aquesta classe conté un mètode abstracte *initializeRules()* que ha de ser implementat a través de classes heretades, per tal d'instanciar el conjunt de regles que defineixen la conversa. Les regles (només aquelles que comencen pel mateix estat que la conversa present) són executades contínuament per la funció d'un intèrpret (mètode *ruleControlFn()*), fins que l'estat final de la conversa és trobat.

Els missatges generats per un agent en el transcurs d'una conversa poden iniciar una nova conversa en un segon agent, de manera que la nova conversa instancia una nova classe de *conversa*.

Cada missatge directe, generat per les converses, porta el seu nom. Tan aviat com un agent comença una conversa, l'afegeix a la llista de converses. En el destinatari, aquesta informació s'interpreta per tal d'encaminar el missatge a l'agent que li correspon.

10.3.3.3. Regles de conversa

Com s'ha vist, les converses estan basades en accions/reaccions que es van succeïnt en un diagrama d'estats. Una vegada definit l'entorn, que és la conversa, cal definir les precondicions dels estats i les accions de cadascun.

Cada regla de conversa descriu les accions que han de ser realitzades quan una conversa es troba en un estat donat. La classe *ConvRule* també és una classe abstracta que l'usuari necessita incloure per a implementar l'aplicació de regles específiques. Si l'estat de la conversa present és el mateix que l'estat actual de la regla i si la conversa és capaç d'executar la regla, llavors el següent estat variable de la regla estableix l'estat de la conversa present (amb el mètode *setPresentConvState()*).

En l'aplicació que s'ha desenvolupat com exemple d'utilització de JAFMAS (v. apartat 10.4), es van realitzar algunes proves d'implementació de converses. Tot i que a la versió definitiva no hi va caler, es van desenvolupar.

Per veure el funcionament/implementació d'una conversa i unes regles, podem veure el següent codi que inicia el *boss* enviant una brossa a un robot.

```
//*****
// File: D:\PFC\Jafmas\tesis_completa\rubfish\RobotBossConv.java
// Conversa entre el Robot i Boss
```

```
import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
```

```
public class RobotBossConv extends Conversation
```

```
{
    private static int convNo = 0;
```

```
    RobotBossConv(Robot agent)
```

```
    {
        //definim l'estat inicial
        super(agent,"start");
        //definim el/s estat/s final/s.
        finalStates.addElement("picked");
        convNo++;
        String name = parent.getAgentName()+"Conv" + convNo;
        setConvName(name);
    }
```

```
    protected void initializeRules()
```

```
    {
        //la conversa en qüestió té dos regles.
        rules.addElement(new RobotConvRule1(this));
        rules.addElement(new RobotConvRule2(this));
    }
```

```
}
```

```
//*****
```

```
class RobotConvRule1 extends ConvRule
```

```
{
```

```
    private RobotBossConv conv = null;
    boolean hihaFeina = false;
    String position;
```

```
    RobotConvRule1(RobotBossConv c) {
        super(c, "cc1","start","working");
        conv = (RobotBossConv)parentConversatn;
    }
```

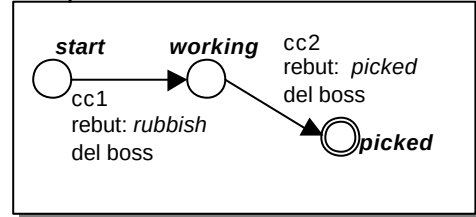
```
    protected void setRecvdMessage()
```

```
    {
        //la conversa s'inicia quan es rep un missatge de tipus rubbish
        transmitMsg = new Message("rubbish");
        position=(String)transmitMsg.getContent();
        hihaFeina=true;
        String nom_robot=(conv.getOwner()).getAgentName();
        (conv.getTextArea()).append("Robot "+nom_robot+" received position new
rubbish");
    }
```

```
    protected void doAfter(){
        //s'ha d'anar a recollir la brossa, i despres confirmar-ho.
        conv.setWait(true);
    }
```

```
}
```

esquema de la conversa *RobotBossConv*



```

//*****

class RobotConvRule2 extends ConvRule
{
    private RobotBossConv conv = null;

    RobotConvRule2(RobotBossConv c) {
        //quan es rep el missatge de "picked" la conversa s'acaba.
        super(c,"cc2","working","picked");
        conv = (RobotBossConv)parentConversatn;
    }

    protected void setRecvdMessage()
    {
        recvdMsg = new Message("picked");
    }

    protected void doAfter(){
        conv.setWait(true);
    }
}

```

10.3.3.4. Encaminament de missatges

Cada agent JAFMAS té un *message router* (implementat a la classe *MsgRouter*). Aquest element de l'agent té la missió d'encaminar els missatges arribats directament a l'agent com a resultat de l'execució de converses amb altres agents.

A més d'encaminar els missatges, el *message router* també té la funció d'iniciar l'execució de noves converses.

El *message router* extreu d'un nou missatge arribat el nom de la conversa a la qual es fa referència. Llavors, mira que aquesta conversa estigui dins de la llista de converses actives de l'agent i, en cas que hi sigui, encua el nou missatge en la cua de missatges de la conversa. Si la conversa a la qual fa referència el nou missatge no està dins la llista de converses obertes, inicia aquesta conversa i afegeix el missatge dins la seva cua.

10.3.4. Nivell multiagent

El nivell multiagent és el nivell més alt de la jerarquia JAFMAS, abans de les interfícies entre l'usuari i l'agent.

Un sistema multiagent fet en JAFMAS comprèn diferents agents que es creen instanciant la classe *Agent*. Aquesta classe comprèn tots els elements que fins ara s'han anat analitzant: transmissió de missatges, manegament de converses, *message router*, etc.

La classe abstracta *Agent* ha de ser extesa per l'usuari segons el domini de l'aplicació. Hi ha diversos mètodes que ja estan implementats (que cal observar-ne la implementació per reescriure'ls si cal) però n'hi ha dos d'abstractes que cal reescriure: *startConversation()* i *addSubjects()*. El primer, *startConversation()* defineix les condicions que s'han de donar per tal que s'iniciïn les diferents converses. El segon, *addSubjects()* dona les adreces (adreça IP) del grups d'agent als que un agent pot subscriure's.

10.3.5. Interfícies d'Agents

JAFMAS proporciona tres interfícies que permeten veure l'estat de diferents activitats de l'agent. Les interfícies són: crear l'agent, monitoritzar l'agent i monitoritzar les converses.

En les subseccions següents es descriuen les característiques bàsiques d'aquests elements, però per veure millor la seva implementació en una aplicació en concret, el lector pot referir-se a l'apartat 1.4. d'aquest capítol.

10.3.5.1. Interfície de creació d'agents

Aquesta interfície és senzilla però força eficient. S'implementa en la classe *CreateAgent*. Ens permet d'inicialitzar diferents paràmetres dels agents.

Permet triar el tipus d'agent, el seu nom, la llista a la qual es vol subscriure inicialment i, l'atribut que es desitgi. Aquests paràmetres són definibles (menys el nom) pel programador i és ell qui ha d'inicialitzar abans les opcions permeses que després podrem escollir. Per inicialitzar aquests paràmetres hi ha uns mètodes que cal inicialitzar adequadament depenent del domini d'aplicació.

Aquests mètodes són:

- *setAgentClassChoice()*: Inicialitzem els tipus d'agents que podrem triar.
- *setSubscribeToList()*: Inicialitzem el nom i les adreces IP dels grups als que podrem subscriure'ns.
- *setAttributeList()*: Inicialitzem els possibles atributs que l'agent pot tenir. Aquests atributs poden variar el significat segons el domini d'aplicació.

10.3.5.2. Interfície de l'agent

Una vegada creat un agent, es genera una interfície per monitoritzar el funcionament de l'agent. Aquesta interfície està implementada en *AgentOpInterface* i *AgentCanvas*.

En la Figura 10.5 podem veure l'aspecte de la interfície. Es dona informació de l'estat de l'agent i es tenen àrees que poden ser usades per donar tot tipus d'informació sobre que està fent l'agent.

Es tenen tres àrees de text que es poden usar segons les necessitats. Inicialment tenen les funcions de mostrar missatges de l'agent, mostrar informació dels missatges que ha rebut directament i, mostrar informació dels missatges que s'han rebut i formen part de converses actives de l'agent.

Aquestes àrees són molt útils i JAFMAS proporciona moltes facilitats per usar-les. Així podem fer:

```
public TextArea dcTextArea;  
dcTextArea=opInterface.getDcTextArea();  
dcTextArea.append((String)missatge_que_volem_mostrar);  
agentTextArea.append((String)missatge_que_volem_mostrar);  
routerTextArea.append((String)missatge_que_volem_mostrar);
```

La variable `agentTextArea` permet accedir a l'àrea de missatges, la variable `dcTextArea` (que, segons les especificacions, hauria d'estar definida, però no ho està) permet accedir a l'àrea de missatges rebuts directament, i `routerTextArea` permet accedir a l'àrea de missatges que rep el router i ha d'encaminar.

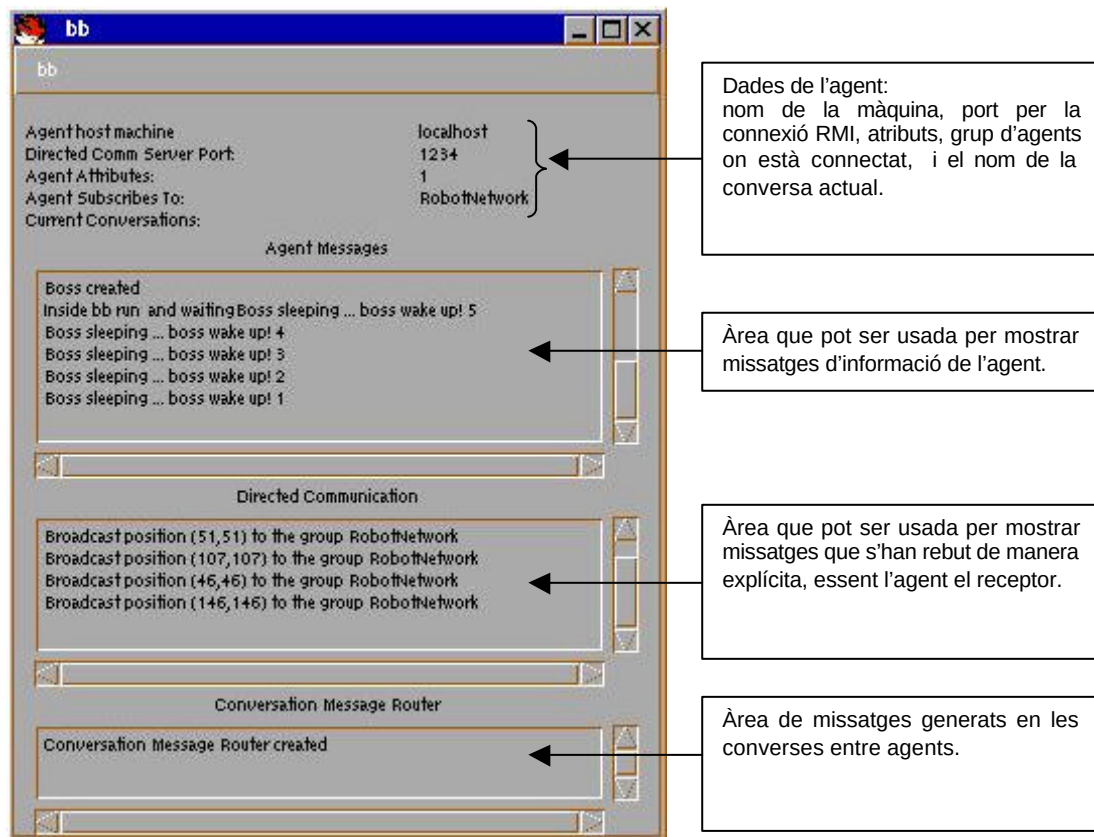


Figura 10.5: Interfície d'un agent destinada a mostrar les seves evolucions/resultats.

10.3.5.3. Interfície de converses

Aquestes interfícies permeten de monitoritzar les diferents transicions que es duen a terme en una conversa. S'implementen en la classe *ConvOpInterface* i *ConvCanvas*. Les interfícies de les converses s'inicien automàticament al mateix temps que s'inicia l'execució d'una conversa.

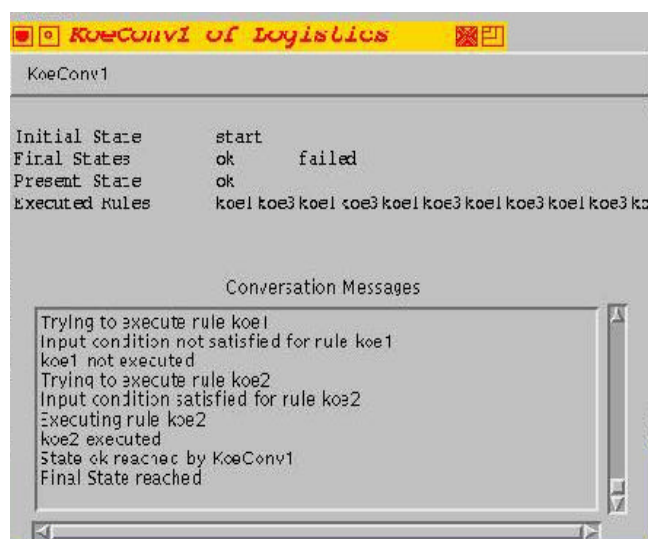


Figura 10.6: Interfície gràfica d'una conversa. Es poden veure tots els missatges que es van generant. A més es mostren l'estat inicial, el/s final/s i l'actual. També es dona informació de les regles executades.

10.4. Desenvolupament d'una aplicació

10.4.1. Descripció de l'aplicació

L'aplicació és la del *món de la brossa*.

En el nostre sistema es tenen dos tipus d'agents: el món i els robots. El món monitoritza l'execució del sistema i va generant periòdicament una brossa en una posició aleatòria. Els robots, que estan esperant, quan detecten aquesta nova brossa, hi van des d'on siguin.

10.4.1.1. Connexions

Quan es crea un agent, l'usuari manté un diàleg amb una interfície de JAFMAS que inicialitza el tipus d'agent que es té, li donem un nom, inicialitzem els seus atributs i també el podem connectar a un grup d'agents (subscriure'l a una llista) (v. Figura 10.7).

Nosaltres no inicialitzem explícitament les connexions d'un agent. Dins el programa cal donar les adreces dels grups als que ens podem connectar i tenir-hi accés. Encara que inicialment només ens podem connectar a un grup, després podem connectar/desconnectar dinàmicament d'altres grups que també s'hagin tingut en compte.

Aquest mètode de treball és útil doncs explota la utilització d'un *communication facilitator* del qual es van veure les seves característiques quan es va parlar del llenguatge KQML (v. p. 3.2.2).

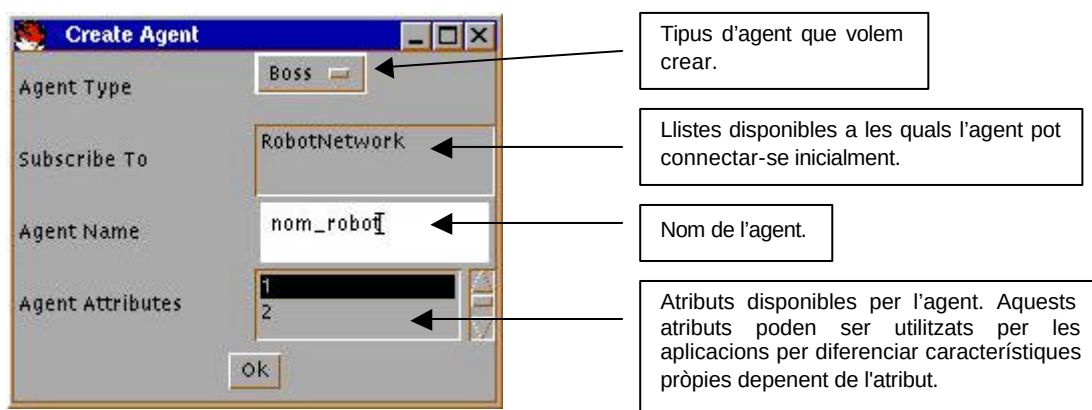


Figura 10.7: Interfície de creació d'un agent.

10.4.1.2. Creació dels agents

Per l'usuari, la creació dels agents és senzilla utilitzant una interfície (v. Figura 10.7). Ara bé, JAFMAS dóna la interfície, però abans d'utilitzar-la cal inicialitzar-la. Per fer-ho, JAFMAS dóna una sèrie de mètodes.

En el desenvolupament de l'aplicació, es va destinar un fitxer (*CreateBossRobots.java*) només per crear/inicialitzar els agents.

Els mètodes per inicialitzar les opcions de la interfície són:

- `setAgentChoice()`: Inicialitza els tipus d'agents disponibles.
- `setSubscribeToList()`: Inicialitza la/es llista/es d'agents a les quals l'agent es pot subscriure.
- `setAttributeList()`: Inicialitzem els atributs disponibles. En la nostra aplicació, el seu contingut no es fa servir, però és una porta oberta per a aplicacions més complexes que necessitin donar unes característiques inicials als agents.

Una vegada inicialitzats tots els camps, cal crear pròpiament l'agent.

La creació d'un agent pot estar determinada pels valors d'alguns dels camps de la interfície. JAFMAS no proporciona mètodes per consultar els continguts i cal utilitzar el valor de les variables que s'implementen a les classes de *CreateAgent*.

Veiem el procés de creació d'un agent:

Fragment de *CreateBossRobots.java*

```
...
CreateBossRobots()
{
    super();
    setAgentChoice();
    setSubscribeToList();
    setAttributeList();
    Font f = new Font("Dialog", Font.PLAIN, 10);
    setFont(f);
    init();
}

protected void createAgent()
{
    if (agentClass.equals("Boss")){
        agent = new Boss(agentName, represents, subscribeTo);
        agent.start();
    }
    if (agentClass.equals("Robot")){
        agent = new Robot(agentName, represents, subscribeTo);
        agent.start();
    }
}

public static void main(String[] args)
{
    Frame f = new CreateBossRobots();
    f.setSize(250, 200);
    f.show();
}
...
```

↑ S'inicialitzen els diferents camps de la interfície. S'executa i esperem que l'usuari ompli totes les opcions.

→ Una vegada confirmat el valor de tots els paràmetres de l'agent, es crea. En aquest cas, segons es tracti d'un *robot* o un *boss* es crea un o altre tipus d'agent.

← Quan s'executa l'aplicació, es crea la interfície.

10.4.1.3. Funcionament dels agents

Els agents són fils d'execució (*threads*) independents. Cadascun dels agents els podem tenir en fitxers diferents, per diferenciar-los. Per implementar el codi d'un agent cal tenir coneixements de Java i llegir atentament la documentació de cadascuna de les classes, perquè cal escriure de nou alguns mètodes i, segons les necessitats de l'aplicació cal reescriure'n d'altres.

El procediment principal per veure com funciona l'agent és `run()`. Veiem aquest mètode per als robots (*Robot*) i per al món (*Boss*).

Fragment de *Boss.java*

```
public void run(){
    //Esperem que es connectin el/s robot/s
    agentTextArea.append("Inside "+agentName + " run ");
    try {
        agentTextArea.append(" and waiting");
        Thread.sleep(5000);
    } catch (InterruptedException e) { }

    while(n_iteracions>0){
        createNewRubbish();
        broadcastMessagePosition(subscribedList,getRubX(),
        getRubY(),n_rub);
        n_rub++;
        try {
            agentTextArea.append("Boss sleeping ... ");
            Thread.sleep(15000);
        } catch (InterruptedException e) { }
        agentTextArea.append("boss wake up! "+n_iteracions+" \n");
        n_iteracions--;
    }
}
```

El boss espera un temps raonable per donar temps a connectar els robots del sistema.

La simulació és per un número d'iteracions màxim. Es crea un nova brossa, s'envia a tots els robots subscrits al facilitador, i dormim 15 segons abans de tornar a començar.

A més hi ha altres funcions que es realitzen a l'agent. L'agent en qualsevol moment pot requerir iniciar alguna conversa amb un altre agent. El mètode `startConversation()` inicia les noves converses de la classe *Conversation* que s'hagin implementat. Altres funcions vindran donades per l'aplicació. Per exemple, en el cas del robot, reescriu el mètode de recepció de missatges via broadcast i, així considera les noves *performatives* que s'implementen en el nostre domini (v. apartat següent –Comunicació–).

```

public void run() {
    iniRobotCoordinate();

    while(!ending){
        try{
            Thread.sleep(100);
            if(new_rubbish_appeared){
                String position=(String)msg_new_rubbish.getContent();
                setRubbish(position);
                working=true;

                int var_ctrl=-1;
                do {
                    var_ctrl=pickingRubbish(xrub,yrub);
                    Thread.sleep(30);
                } while ((var_ctrl!=1) && (var_ctrl!=2));

                if (var_ctrl==1)
                    broadcastMessagePicked(subscribedList,position);
                if (var_ctrl==2)
                    agentTextArea.append("Rubbish picked by another Robot \n");

                new_rubbish_appeared=false;
                agentTextArea.append(agentName+": Rubbish picked up!\n");
            }
        } catch (InterruptedException e) {}
    }
}

```

Quan s'executa el mètode per primera vegada, el robot es situa dins l'àrea permesa aleatòriament.

Igual que succeïa amb JATLite, el thread de l'agent *no pot acaparar* tots els recursos. Per això es mostreja/actualitza l'agent periòdicament.

La detecció d'una nova brossa es fa automàticament quan es rep una *performative* del tipus *new_rubbish* enviada pel món.

Cada robot només recull una brossa alhora, i fins que no la recull o li diuen que s'aturi, no deixa aquest objectiu.

El robot va movent-se per l'àrea permesa en moviments d'un salt cada cop. Podria variar-se la velocitat, variant el temps entre salts.

La recol·lecció d'una brossa pot acabar-se de dues maneres: si *var_ctrl=1* la brossa ha estat recollida pel robot; si és 2, un altre agent se li ha avançat.

Una vegada s'ha acabat amb una brossa, s'inicialitza l'entorn per rebre més tasques.

10.4.1.4. Comunicació

La comunicació entre els agents es realitza a través de missatges. En aquest cas, només s'utilitza el mètode de difusió o *broadcast*.

En la nostra aplicació aquest mètode funciona molt bé. L'agent *boss* cada cop que crea una brossa, difon la seva posició a *tots* els robots que hi hagi al sistema connectats. No li importa si en són un o tretze, doncs són ells qui s'han de barallar per anar a buscar-la. Els robots, per altra banda, utilitzen aquest mètode per avisar quan han recollit una brossa a la resta de robots.

El llenguatge de comunicació entre agents que implementa JAFMAS és KQML. Com ja s'ha vist, aquest llenguatge diferencia entre diferents tipus d'accions. Es poden utilitzar les que el sistema proporciona, però pel nostre cas, s'han definit dues de noves.

Quan es crea una nova brossa, el món envia una *performative* de tipus *new_position* indicant la seva posició. Els robots avisen que una brossa ha estat recollida enviant una *performative* de tipus *picked* i la posició de la brossa recollida.

També es podia haver enviat una *performative* de tipus *tell* i tractar el seu contingut.

JAFMAS dedica una atenció especial a la comunicació i els missatges transmesos. La interfície dels agents té àrees especialment dedicades per mostrar missatges que hi facin referència. En la nostra aplicació s'han utilitzat aquestes àrees per mostrar molts missatges i, així controlar bé les evolucions de la simulació.

10.4.1.5. Resultats obtinguts

Els resultats obtinguts i la forma de treballar de JAFMAS poden considerar-se molt satisfactoris. Com s'ha vist en els apartats precedents, JAFMAS posa molta atenció en les interfícies gràfiques que permeten un control de l'execució de l'aplicació òptim.

A més, el codi font de les classes està disponible i pot completar-se/canviar-se segons les necessitats del programador. En el nostre cas, s'han canviat algunes característiques de la presentació de la interfície de l'agent per adequar-les millor al nostre sistema.

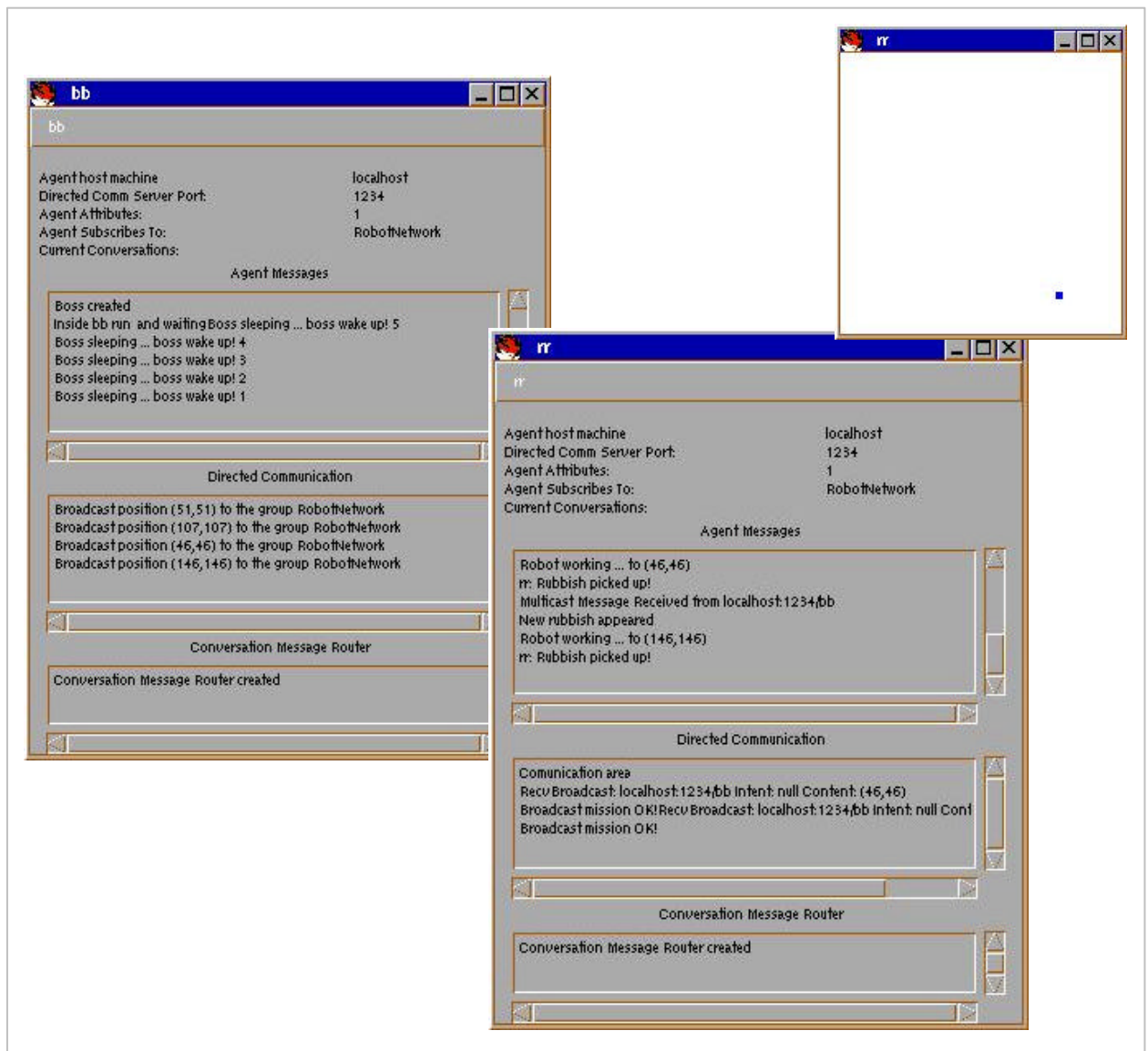


Figura 10.8: Interfícies gràfiques dels agents món (*bb*) i robot (*rr*) i, el tauler del robot.

10.4.2. Codi complet

10.4.2.1. Creació dels agents

```
//*****
// File: D:\PFC\Jafmas\tesis_completa\rubbish\CreateBossRobots.java
// Class to create Agents

import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.net.*;

public class CreateBossRobots extends CreateAgent
{
    protected void setAgentChoice(){
        agentClassChoice.addItem("Boss");
        agentClassChoice.addItem("Robot");
    }

    protected void setSubscribeToList(){
        subscribeToList.add("RobotNetwork");
    }

    protected void setAttributeList(){
        attributeList.add("1");
        attributeList.add("2");
        attributeList.add("3");
        attributeList.add("4");
    }

    CreateBossRobots()
    {
        super();
        setAgentChoice();
        setSubscribeToList();
        setAttributeList();
        Font f = new Font("Dialog",Font.PLAIN,10);
        setFont(f);
        init();
    }

    protected void createAgent()
    {
        if (agentClass.equals("Boss")){
            agent = new Boss(agentName,represents,subscribeTo);
            agent.start();
        }
        if (agentClass.equals("Robot")){
            agent = new Robot(agentName,represents,subscribeTo);
            agent.start();
        }
    }

    public static void main(String[] args)
    {
        Frame f = new CreateBossRobots();
        f.setSize(250,200);
        f.show();
    }
}
```

10.4.2.2. Agent Boss

```
//*****
// File: D:\PFC\Jafmas\tesis_completa\rubbish\Boss.java
// AGENT : BOSS

import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.net.*;
import java.rmi.*;

class Boss extends Agent
{
    public Conversation conv = null;
    String subscribedList;
    int ruby=0;
    int rubx=0;
    int n_iteracions=5;
    public TextArea dcTextArea;
    boolean inici_conversa=false;
    String r0 = "hola";
    Message msg_escollit;
    Vector msg_recvd;
    int n_rub=0;

    Boss(String name,String[] rep, String[] subjectList){
        super(name,rep,subjectList);
        //thisColumn = Integer.parseInt(rep[0]);
        //addResourceReqd();
        //bossOp = new BossOpInterface(this);
        //bossOp.setSize(200,200);
        //bossOp.show();
        dcTextArea=opInterface.getDcTextArea();
        agentTextArea.append("Boss created \n");
        //advertisePresence(subjectList);
        subscribedList=subjectList[0];
        msg_recvd=new Vector();
    }

    void broadcastMessagePosition(String list, int x, int y, int n){
        String content = "(" + x + "," + y + ")";
        dcTextArea.append("Broadcast position "+content+" to the group
"+subscribedList+"\n");
        Message msg = new Message("new_position");
        msg.setSender(getFullName());
        msg.setContent((String)content);
        String intent=r0+n;
        msg.setIntent(intent);
        MulticastCom mcCom = getMcCom(subscribedList);
        mcCom.sendMessage(msg);
    }

    //PRIVATE METHODS
    public int generateRandomInt(int n){
        Random rand = new Random ((long)System.currentTimeMillis());
        int a = Math.abs(rand.nextInt()%n);
        return a;
    }

    private void createNewRubbish(){
        int n=190;
        setRubX(generateRandomInt(n));
        setRubY(generateRandomInt(n));
    }
}
```

```

    }

    public void run(){
        //Esperem que es connectin el/s robot/s
        agentTextArea.append("Inside "+agentName + " run ");
        try {
            agentTextArea.append(" and waiting");
            Thread.sleep(5000);
        } catch (InterruptedException e) { }

        while(n_iteracions>0){
            createNewRubbish();
            broadcastMessagePosition(subscribedList,getRubX(),
getRubY(),n_rub);
            n_rub++;
            try {
                agentTextArea.append("Boss sleeping ... ");
                Thread.sleep(15000);
            } catch (InterruptedException e) { }
            agentTextArea.append("boss wake up! "+n_iteracions+" \n");
            n_iteracions--;
        }
    }

    public void addSubjects()
    {
        subjectList.addElement("QueenNetwork");
        subjectAddrList.addElement("127.0.0.1");
    }

    public void startConversation(Object m)
    {

    }

    public int getRubX() {return rubx;}
    public int getRubY() {return ruby;}
    public void setRubX(int p) {rubx=p;}
    public void setRubY(int p) {ruby=p;}

    public void setMsgRobotChoiced(Message m){
        msg_escollit=m;
    }
    public Message getMsgRobotChoiced(){
        return msg_escollit;
    }
}

```

10.4.2.3. Agent Robot

```
//*****
// File: /mnt/d/pfc/Jafmas/tesis_completa/rubbish/Robot.java
// AGENT : Robot

import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.net.*;
import java.rmi.*;

class Robot extends Agent
{
    int thisColumn = 1;
    public int posX = 1;
    public int posY = 1;
    public int xrub,yrub;
    public int i,j;
    private boolean tenirFeina = false;
    public RobotOpInterface robotOp = null;
    public static final int totalQueens = 4;
    public boolean new_rubbish_appeared = false;
    public Message msg_new_rubbish=null;
    public boolean ending = false;
    public boolean working = false;
    public String position="";
    TextArea dcTextArea;
    String subscribedList;

    Robot(String name,String[] rep, String[] subjectList) {
        super(name,rep,subjectList);
        thisColumn = Integer.parseInt(represents[0]);
        //addResourceReqd();
        robotOp = new RobotOpInterface(this);    //creeem el tauler on
veurem les evolucions
        robotOp.show();
        agentTextArea.append("Agent robot ["+name+"] created \n");
        dcTextArea=opInterface.getDcTextArea();
        dcTextArea.append("Comunication area\n");
        subscribedList=subjectList[0];
    }

    public void addSubjects()
    {
        subjectList.addElement("RobotNetwork");
        subjectAddrList.addElement("127.0.0.1");
    }

    //Reescribim el metode
    public void setRecvdMulticastMsg(Message msg)
    {
        msg.setRecvTime();
        agentTextArea.append("Multicast      Message      Received      from
"+msg.getSender()+"\n");
        if ("require".equals(msg.getMsgType())){
            requestedResrcProvider.respond(msg);
        }
        if ("new_position".equals(msg.getMsgType())){
            msg_new_rubbish=(Message)msg;
            agentTextArea.append("New rubbish appeared\n");
            dcTextArea.append("RecvBroadcast: "+msg.getSender());
            dcTextArea.append(" Intent: "+msg.getIntent());
            dcTextArea.append(" Content: "+msg.getContent()+" \n");
            //caldra recollir la brossa de la posicio.
        }
    }
}
```

```

        new_rubbish_appeared=true;
    }

    if ("picked".equals(msg.getMsgType()))                &&
    !(msg.getSender().equals(getFullName())) {
        agentTextArea.append("Rubbish picked by
"+msg.getSender()+"\n");
        stopToPick();
    }

    if ("advertise".equals(msg.getMsgType())){
        //new_rubbish_appeared=true;
        //agentTextArea.append("New rubbish appeared\n");
        //cal contestar amb la nostra posicio.
    }
}

public void broadcastMessagePicked(String subscribedList, String
position){
    dcTextArea.append("Broadcast mission OK!");
    Message msg = new Message("picked");
    msg.setSender(getFullName());
    msg.setContent((String)position);
    MulticastCom mcCom = getMcCom(subscribedList);
    mcCom.sendMsg(msg);
}

public void sendResponseRubbish(Message m,int x, int y){
    System.out.println("Envio posicio al boss");
    //agentTextArea.append("Trying responding to
"+m.getSender()+"\n");
    String pos = "("+x+","+y+")";
    Message msg = new Message("position");
    msg.setSender(getFullName());
    msg.setContent((String)pos);
    msg.setReceiver(m.getSender());
    msg.setIntent(m.getIntent());
    directedComImpl.sendMsg(msg);
    dcTextArea.append("Sended "+pos+" to "+msg.getReceiver());
}

private void setRubbish(String position){
    agentTextArea.append("Robot working ... to "+position+"\n");
    String numstr="";
    int n[] = new int[4];
    int strlen = position.length();
    int j=0;
    for(int i=0;i<strlen;i++) {
        char c = position.charAt(i);
        if(c==' '){
            n[j]= Integer.parseInt(numstr);
            numstr="";
            j++;
        }
        if( (c!='\r') && (c!='\n') && (c!='(') && (c!='\n') && (c!='\n') && (c!='\n') ) {
            numstr=numstr+c;
        }
    }
    n[j]=Integer.parseInt(numstr);
    xrub=n[0];
    yrub=n[1];
}

public void moveRobot(int a, int b) {
    setRobotX(a);
    setRobotY(b);
}

```

```

}

public void stopToPick(){
    working=false;
}

public int pickingRubbish(int x1, int y1){
    int x=getRobotX();
    int y=getRobotY();
    int dx=x1-x;
    int dy=y1-y;
    int absdx=Math.abs(dx);
    int absdy=Math.abs(dy);

    if (working) {
        if ((dx==0) && (dy==0)) { return 1; }
        else if (dx==0)
            moveRobot(x, y+(dy/absdy) );
        else if (dy==0)
            moveRobot(x+(dx/absdx), y);
        else if (absdx>=absdy)
            moveRobot(x,y+(dy/absdy));
    }
    else
    {
        return 2;
    }

    //robotOp.pinta2(xrub,yrub,getRobotX(),getRobotY());
    int tipus=3;
    robotOp.pinta(tipus,getRobotX(),getRobotY());
    robotOp.repaint();
    return 0;
}

private int generateRandomInt(int n){
    Random rand = new Random ((long)System.currentTimeMillis());
    int a = Math.abs(rand.nextInt()%n);
    return a;
}

private void iniRobotCoordinate(){
    int MAX=200;
    int tipus=3;

    int x=generateRandomInt(MAX);
    setRobotX(x);
    int y=generateRandomInt(MAX);
    setRobotY(y);

    robotOp.pinta(tipus,x,y);
    robotOp.repaint();
}

public void run() {
    iniRobotCoordinate();

    while(!ending){
        try{
            Thread.sleep(100);
            if(new_rubbish_appeared){
                String position=(String)msg_new_rubbish.getContent();
                setRubbish(position);
                working=true;
            }
        }
    }
}

```

```

        int var_ctrl=-1;
        do {
            var_ctrl=pickingRubbish(xrub,yrub);
            Thread.sleep(30);
        } while ((var_ctrl!=1) && (var_ctrl!=2));

        if (var_ctrl==1)
            broadcastMessagePicked(subscribedList,position);
        if (var_ctrl==2)
            agentTextArea.append("Rubbish    picked    by    another
Robot \n");

            new_rubbish_appeared=false;
            agentTextArea.append(agentName+": Rubbish picked up!\n");
        }
    } catch (InterruptedException e) {}
}

public void startConversation(Object m){

}

// ACCESSOR METHODS

public int getRobotX(){return posx;}
public int getRobotY(){return posy;}

public void setRobotX(int p){posx=p;}
public void setRobotY(int p){posy=p;}

public String getPositionRubbish() {return position;}

}

```

10.4.2.4. Tauler del robot

```
//*****
// File: D:\PFC\Jafmas\tesis_completa\rubbish\RobotOpInterface.java
// ROBOT POSITION OPERATOR INTERFACE

import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.net.*;

class RobotOpInterface extends Frame
{
    Robot parent = null;
    int posX=100;
    int posy=100;
    int posX1;
    int posy1;
    int tipusFigura=1;
    int tipusFigura1=-1;

    RobotOpInterface(Robot b)
    {
        parent = b;
        setTitle(b.getAgentName());
        setBackground(Color.white);
        setForeground(Color.blue);
        setSize(207,220);
        Font f = new Font("Helvetica",Font.PLAIN,10);
        setFont(f);
    }

    public void redrawCanvas(){
        repaint();
    }

    public void pinta(int tipus, int x1, int y1) {
        tipusFigura=tipus;
        posX=x1;
        posy=y1;
        repaint();
    }

    public void pinta2(int x1, int y1, int x2, int y2) {
        tipusFigura1=1;
        posX=x1;
        posy=y1;
        posX1=x2;
        posy1=y2;
        repaint();
    }

    public void paint(Graphics g) {
        posX=posx+5;
        posy=posy+20;

        if (tipusFigura1==1) {
            tipusFigura1=-1;
            g.setColor(Color.black);
            g.fillRect(posx1+5,posy1+20,5,5);
            g.setColor(Color.magenta);
            g.drawRect(posx,posy,5,5);
        }
        else if (tipusFigura==1) {
            g.setColor(Color.magenta);
        }
    }
}
```

```
        g.drawRect(posx, posy, 5, 5);
    }
    else if (tipusFigura==2)
    {
        g.setColor(Color.black);
        g.fillRect(posx, posy, 5, 5);
    }
    else if (tipusFigura==3)
    {
        g.setColor(Color.blue);
        g.fillRect(posx, posy, 5, 5);
    }
}
}
```

10.5. Aplicacions pràctiques

Una de les aplicacions que es difícil d'implementar és la d'aplicacions que treballen sobre dades d'Internet, com per exemple agents d'informació, o aplicacions sobre WWW, perquè no es proporciona cap eina per treballar-hi.

10.6. Conclusions

JAFMAS és una eina de treball excel·lent. Si fos la primera eina de desenvolupament que hom veiés i estudiés, potser no veuria totes les coses bones que té.

JAFMAS no només dona una metodologia de treball, sinó que a més proporciona un completíssim conjunt de classes que permeten crear un sistema multiagent complet. Crea una classe abstracte *agent* i permet de ser configurada segons les necessitats de l'aplicació; com es va veure, amb JATLite és el programador qui crea l'agent.

Aquest fet es pot veure del costat negatiu, doncs potser al programador no li interessa la forma de les interfícies i/o la implementació interna del codi. Com que es proporciona el codi font de les classes, es pot provar de canviar segons les necessitats.

Una avantatge important és la reutilització de codi. Aquesta reutilització pot ser de dos tipus: heretat o reusar codi Java. Java permet de desenvolupar eines que embolcallin a codis escrits altres llenguatges com C o Lisp, i crear un nou agent Java; ara bé, podem utilitzar interfícies o programes escrits en Java que ja tinguem, per exemple, podem interactuar amb bases de dades a través de JDBC.

La distribució de JAFMAS és lliure, i senzill d'instal·lar.

11. ESTUDI DE ZEUS

ZEUS

British Telecom
(Intelligent Systems Research Group)
Jaron Collins, Divine Ndumu, [et. al]

wysiwyg://59/http://www.labs.bt.com/projects/agents/zeus/index.html



David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

11. ESTUDI DE ZEUS	155
11.1. DESCRIPCIÓ GENERAL.....	157
11.2. METODOLOGIA.....	159
11.2.1. <i>Nivells d'abstracció</i>	159
11.2.2. <i>Procés de desenvolupament</i>	162
11.2.2.1. Ontologia.....	162
11.2.2.2. Creació dels agents.....	164
11.2.2.3. Configuració de les utilitats dels agents.....	172
11.2.2.4. Configuració de les tasques de l'agent.....	174
11.2.2.5. Implementació de l'agent	175
11.3. ARQUITECTURA.....	176
11.3.1. <i>Entorn de desenvolupament</i>	176
11.3.1.1. Llibreria de components d'un agent (Agent Component Library)	176
11.3.1.2. Software de construcció (Agent Building Software)	178
11.3.1.3. Utilitats dels agents (ZEUS Utility Agents).....	179
11.3.2. <i>Descripció d'un agent</i>	182
11.4. DESENVOLUPAMENT D'UNA APLICACIÓ.....	184
11.4.1. <i>Descripció de l'aplicació</i>	184
11.4.1.1. Inici del sistema	184
11.4.1.2. Creació dels agents.....	188
11.4.1.3. Funcionament dels agents.....	192
11.4.1.4. Comunicació	193
11.4.1.5. Resultats obtinguts	197
11.4.2. <i>Codi de l'aplicació</i>	200
11.4.2.1. Interfície WorldAgent	200
11.4.2.2. Interfície Robot	200
11.4.2.3. Brossa.....	201
11.5. APLICACIONS PRÀCTIQUES I CONCLUSIONS.....	210

11.1. Descripció general

La dificultat innata de construir sistemes multiagent ha motivat als investigadors d'agents a moure's per diversos camps que desenvolupin solucions, noves metodologies i eines per a construir-los. Aquesta és també la filosofia que segueix el desenvolupament de les eines de construcció d'agents ZEUS (*Agent Building Toolkit*), el qual facilita el ràpid desenvolupament de les aplicacions dels agents col·laboratius a través de components de diferents nivells i un entorn que suporti el procés de construcció. Les eines ZEUS són un conjunt de noves tecnologies amb solucions que proporcionen la construcció d'un entorn integrat a l'agent.

Zeus proporciona un conjunt de components, escrits en el llenguatge de programació Java, el qual suporta el desenvolupament dels sistemes multiagent. Hi ha llibreries dels components d'un agent, eines de construcció d'agents i un seguit d'utilitats força ampli.

La llibreria de components dels agents (*Agent Component Library*) és una col·lecció de classes que configuren els blocs de construcció dels agents individuals. Totes aquestes classes juntes implementen l'aplicació així com diferents aspectes de la funcionalitat dels agents, tals com comunicació, ontologia i interacció social.

Amb ZEUS, una aplicació específica d'un agent pot ser construïda especialitzant l'agent genèric ZEUS, és a dir, es proporciona l'entorn i nosaltres modelem l'agent. Per tal de facilitar el ràpid desenvolupament de sistemes multiagent, les eines ZEUS proporcionen un entorn de desenvolupament visual que suporta un plantejament d'agents d'alt nivell. Utilitzant aquest plantejament, l'investigador de sistemes multiagent no necessita conèixer els detalls de la llibreria de components dels agents sinó conèixer les eines de les quals pot fer ús.

Al nivell més alt d'abstracció, el desenvolupament d'agents ZEUS requereix veure als agents formats per tres nivells: el nivell de definició, el nivell d'organització i el nivell de coordinació.

En el nivell de definició, l'agent és vist com una entitat de raonament autònom, en termes de competència, model racional, recursos, creences, preferències, etc. El nivell d'organització és vist en termes de les relacions amb altres agents, com estan informats i quines habilitats tenen. Al nivell de coordinació, l'agent és vist com una entitat social, en termes de coordinació i de tècniques de negociació.

Aquest model d'agent és complementat amb els protocols, que implementen la comunicació entre agents, i una aplicació d'interfície, que permet generar el codi dels agents, que són lincats per programes externs (en aquest cas es fa servir Java). Alhora, s'ha pensat en un sistema dinàmic i que pot adequar-se a moltes necessitats; per això s'han deixat portes obertes per la inclusió de recursos externs al sistema Zeus, com per exemple programes dissenyats per a ser usats pels agents d'una aplicació determinada, la utilització de Bases de Dades preexistents, etc.

El sistema Zeus és un entorn de desenvolupament. Es donen una sèrie d'eines, opcions, paràmetres, recursos, ... que cal saber utilitzar. Per utilitzar el sistema correctament, Zeus dóna una metodologia de desenvolupament per analitzar els dominis d'aplicació, i traduir tota la informació rellevant del món real a informació que es farà servir per desenvolupar l'aplicació en Zeus (igual a qualsevol tipus de sistema).

El producte estudiat és una versió d'avaluació del producte per la qual es va demanar permís i es va signar un contracte entre la Universitat Rovira i Virgili i British Telecom. Junt amb la llicència es pot baixar una extensa documentació de recent creació. Encara no està completa en la seva totalitat, però s'inclou un manual de referència extens i la documentació de dos exemples implementats. En un futur es publicarà la documentació d'un tercer exemple molt més extens que els dos anteriors.

Un handicap important d'aquest entorn és la utilització de l'últim compilador de Java que ha sortit al mercat, el JDK 1.2.. Tot i que ja fa un temps que va sortir i que la versió per Windows està molt estesa, la versió per Linux no ho està tant i pot costar d'obtenir. Tot i això pot obtenir-se més informació a la pàgina web de Sun: <http://java.sun.com>.

11.2. Metodologia

Els desenvolupaments d'agents ZEUS s'inicien per un estadi d'anàlisi, on el problema és investigat i modelat. La metodologia que segueix no prescriu cap plantejament particular a l'anàlisi del problema, deixant llavors als investigadors la llibertat d'escollir els seus propis plantejaments. El plantejament recomanat és el *Role Modelling*, que encoratja a l'investigador a pensar el problema en termes de papers/tasques, que necessiten ser interpretats, i de les responsabilitats i col·laboracions associades a cada paper.

11.2.1. Nivells d'abstracció

Usant una tècnica com la de *Role Modelling* per primera vegada, el disseny del procés comença per donar a conèixer a l'usuari quins agents haurien de ser-hi presents, i quines responsabilitats han de satisfer. Aleshores, el procés inclou el trasllat de responsabilitats al nivell d'agents quan es presenten problemes i donar les solucions més apropiades.

Amb dissenys conceptuals produïts per cada agent, l'eina ZEUS pot ser usada per a realitzar implementacions. El procés consisteix en diversos estadis, els quals estan estretament enllaçats als nivells d'abstracció que existeixen en un agent de ZEUS. Així, al nivell més alt d'abstracció, un disseny d'un agent ZEUS és vist com una entitat, tal com s'il·lustra a la Figura 11.1.

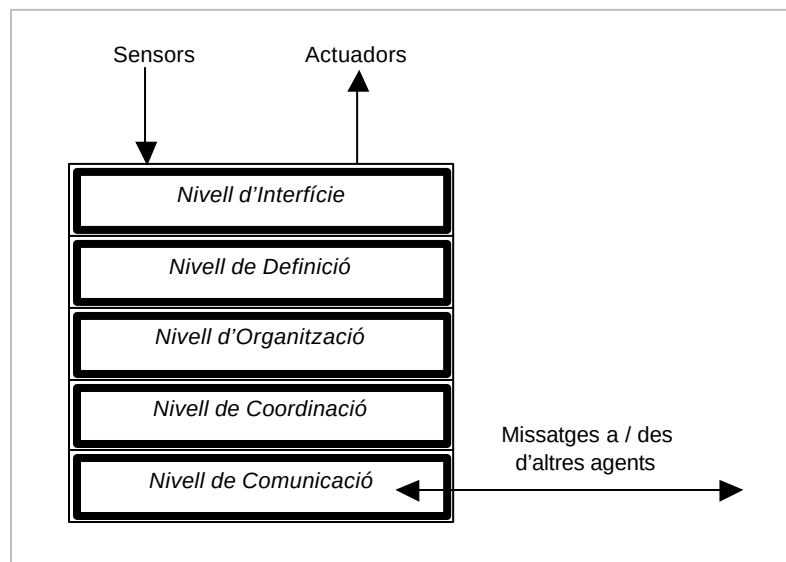


Figura 11.1: Nivells d'abstracció de la metodologia Zeus.

Bàsicament, les funcions de cada nivell són:

- ➔ *Nivell d'interfície*, el qual permet a l'agent ser compilat junt amb programes externs que proporcionen un pont entre l'usuari i el sistema.
- ➔ *Nivell de Definició*, on l'agent és vist com una entitat de raonament autònoma.
- ➔ *Nivell d'Organització*, on l'agent és vist en termes de relacions amb altres agents.
- ➔ *Nivell de Coordinació*, on l'agent és vist com una entitat social que interactua d'acord als protocols coneguts i estratègies.
- ➔ *Nivell de Comunicació*, el qual implementa els protocols i mecanismes que suporten la comunicació entre agents.

El *nivell d'interfície* rep entrades a través de sensors i és capaç de canviar el món extern a través dels seus actuadors. Com els sensors i els actuadors són externs a l'agent, la configuració del *nivell d'interfície* consisteix en especificar interfícies entre ells i els agents. Com es veu en l'aplicació d'exemple dissenyada en Zeus, aquest nivell l'hem implementat en un programa extern, on és l'usuari qui *actua* per iniciar la simulació. Però després és l'agent principal qui *actua* en altres agents per a dur a terme una acció.

Al *nivell de definició*, l'agent és una entitat de raonament en termes d'habilitats, objectius, creences i preferències. Com aquest nivell és físicament format per classes de la llibreria de Components de ZEUS (*ZEUS Agent Component Library*), aquest estadi de la metodologia inclou la configuració d'aquests components.

Al *nivell d'organització*, l'agent configura les relacions que ha de mantenir amb la resta d'agents de la comunitat. S'introdueix el concepte d'*agency* (grup d'agents relacionats). Aquests grups poden ser reals i relacionats per atributs comuns, tal com si formessin part de la mateixa companyia. També poden ser virtuals, llavors els seus constituents comparteixen un acord de cooperació. En aquest nivell la metodologia suposa configurar l'agent en termes de l'*agency* al qual pertany, quins papers juga en el grup, de quins altres agents està informat i quines habilitats coneix dels altres agents. Aquestes relacions influeixen en el funcionament de la comunitat d'agents. Es poden tenir relacions de submissió (un agent està sotmès a un altre), de paritat (dos agents són iguals) o de col·laborador (en cas d'haver de realitzar alguna tasca, es pot requerir l'ajuda de col·laboradors per a dur-la a terme).

Al *nivell de coordinació*, l'agent es considera una entitat social; això implica configurar-lo segons el comportament social que s'hagi de dur a terme. S'aconsegueix mitjançant l'equipament dels protocols de negociació apropiats i noves estratègies. Zeus dóna una importància especial a aquestes estratègies de negociació, i es proporciona una interfície específica per a implementar-les. La negociació està pensada en termes de costos, preferències, i temps d'execució; i les estratègies poden ser configurades segons les necessitats que es tinguin. A més, es té l'opció de crear noves estratègies de negociació de forma externa al sistema Zeus, i que després es poden integrar. També es proporciona coordinació en l'execució de les tasques dels agents.

El *nivell de comunicació*, amb la base d'un protocol de transport, proporciona un llenguatge que permet als agents enviar missatges als altres. Per compatibilitat, aquest nivell hauria de ser el mateix per a cada agent; no és necessari que els usuaris configurin aquest nivell. El llenguatge de comunicació està basat en KQML i per tant es tenen una sèrie d'accions permeses o performatives. Es basa en KQML però no és exactament igual que l'especificació de l'estàndard, perquè s'inclouen alguns camps per optimitzar el rendiment del sistema.

Un agent doncs és configurat com a membre d'una societat multiagent. El procés de configuració s'acompleix utilitzant l'eina *ZEUS Agent Generator* (v.Figura 11.2), la qual al final genera el codi font. S'han d'especificar els recursos que es posseeixen a l'inici i les tasques que són capaços de fer. L'usuari només necessita implementar la funcionalitat i aplicació específica de l'agent, i usant les interfícies proporcionades es genera el codi per compilar.

Una vegada compilats els codis font generats, es pot dur a terme l'execució de l'aplicació.



Figura 11.2: Interfície gràfica de ZEUS Agent Generator.

11.2.2. Procés de desenvolupament

L'objectiu del procés és crear implementacions de treball dels agents a partir de dissenys conceptuals creats a l'etapa prèvia. Consta de diversos estadis (alguns dels quals són derivats dels nivells d'abstracció mostrats a la Figura 11.1.

ETAPES:

Pas 1	Creació de l'Ontologia.
Pas 2	Creació de l'Agent.
Pas 3	Configuració de les Utilitats d'un agent.
Pas 4	Configuració de les tasques de l'Agent.
Pas 5	Implementació de l'agent.

11.2.2.1. Ontologia

Abans d'implementar cap agent, l'usuari ha de definir l'*ontologia de l'aplicació*. L'ontologia d'una aplicació és el coneixement que representa els conceptes significants dins del domini d'aplicació particular.

L'eina usada per tal d'entrar aquesta informació és el *ZEUS Ontology Editor* (v.Figura 11.4).

La importància del concepte és fàcilment avaluada; la interacció entre agents no es pot donar sense que ambdues bandes en siguin conscients i entenguin el que s'estan dient. L'ontologia dóna aquesta base en la qual els agents es desenvolupen i interaccionen entre ells.

Abans de començar aquesta etapa de desenvolupament, s'han d'haver identificat:

- Els conceptes clau dins del domini del problema.
- Els atributs significants de cada concepte.
- Els tipus de cada atribut.
- Qualsevol limitació dels atributs.

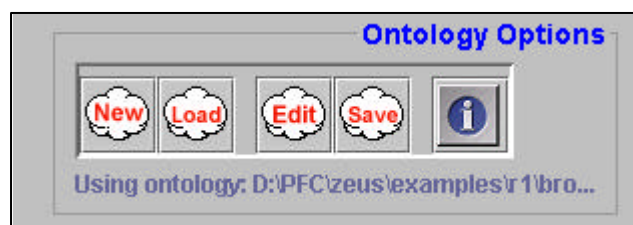


Figura 11.3: Opcions d'edició d'una ontologia.

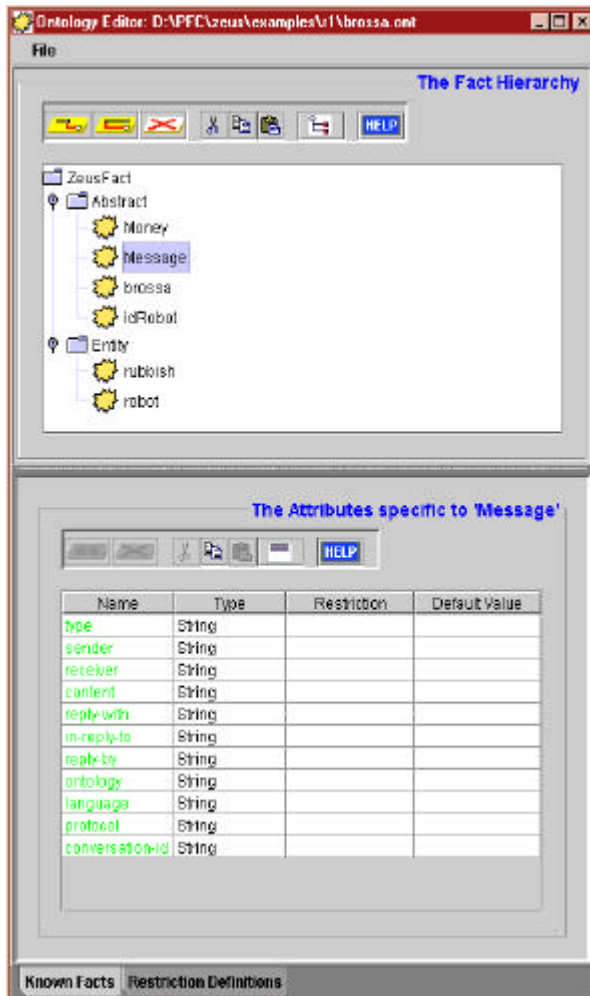


Figura 11.4: Detall de l'editor de l'ontologia. En aquest cas veiem detallat el tipus *message* definit per defecte per Zeus. Es poden veure els diferents camps que s'han definit.

L'editor de l'ontologia permet de crear els conceptes que es creuen importants per la utilització en el sistema d'agents de l'aplicació que definim. Es poden definir dos tipus de conceptes: fets (*Abstract*) i variables (*Entity*). Els primers són els elements que es poden transmetre dins de missatges i poden referir-se en les regles de comportament (tipus especial de tasques); les variables són conceptes que es fan servir en un altre tipus de tasques (primitives), i es poden instanciar com recursos de l'agent.

Els fets i variables estan constituïts per diferents camps, i aquests són de diferents tipus. Els tipus que es poden definir són: *String*, *Integer*, *real*, *boolean*, *date*, *time*, i *JavaObject*. Alhora es poden determinar restriccions sobre el valor dels camps i donar un valor per defecte, si cal.

11.2.2.2. Creació dels agents

Durant aquest estadi, l'agent ZEUS genèric és configurat per a satisfer les seves responsabilitats en una aplicació específica. S'han de prendre una sèrie de decisions:

- Quins agents existeixen?
- Quines activitats ha de realitzar cada agent?
- Com interactuarà cada agent amb la resta?
- Quines estratègies i aptituds coneix cada agent d'ell mateix i d'altres agents?

Aquest procés de creació suposa al *ZEUS Agent Editor* completar quatre subetapes (depenent de la naturalesa de l'agent): *Descripció de l'agent*, *descripció/disseny de tasques*, *organització de l'agent*, i *coordinació de l'agent*.

DESCRIPCIÓ DE L'AGENT

En aquesta etapa s'especifiquen les tasques, recursos inicials i planning d'habilitats de cada agent. Es tracta de pensar que farà i com es comportarà l'agent.

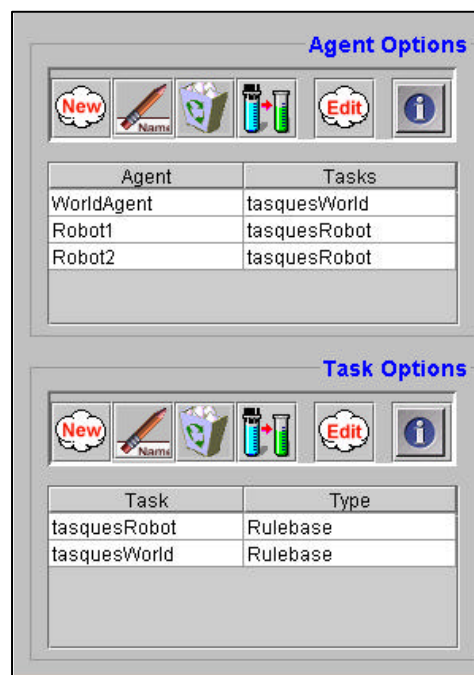


Figura 11.5: Part de *ZEUS Agent Generator* destinada a la creació dels agents. Es descriuen els agents de l'aplicació i les tasques que duen a terme. Mes enllà, cal editar cada agent per separat i configurar-lo adequadament.

DESCRIPCIÓ/DISENY DE TASQUES

En la descripció de tasques s'especifica l'aplicabilitat i atributs de les activitats de l'agent. Les tasques que portarà a terme cada agent cal definir-les prèviament en la interfície destinada a tal efecte *Task Options* (v. Figura 11.6).

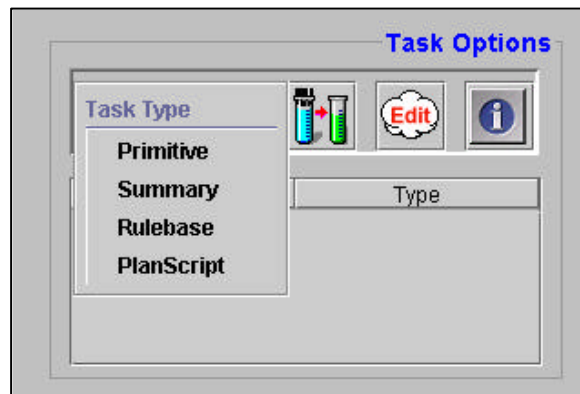


Figura 11.6: Detall dels diferents tipus de tasques que es poden definir en *ZEUS Agent Generator*.

Les tasques que pot dur a terme un agent poden ser de quatre tipus diferents: *primitive*, *summary*, *rulebase* i *planscript*.

Les primitives estan enfocades en termes de precondicions, efectes, cost, durada, restriccions, i ordre entre precondicions, mentre que tasques *summary* o *planscript* estan definides en termes d'un mini-pla que està format de primitives. Les regles de comportament emulen el funcionament d'un sistema expert, on les regles van comandant el funcionament del sistema. En aquest cas les regles no comanden el funcionament, sinó que s'integren dins el sistema complementant la resta.

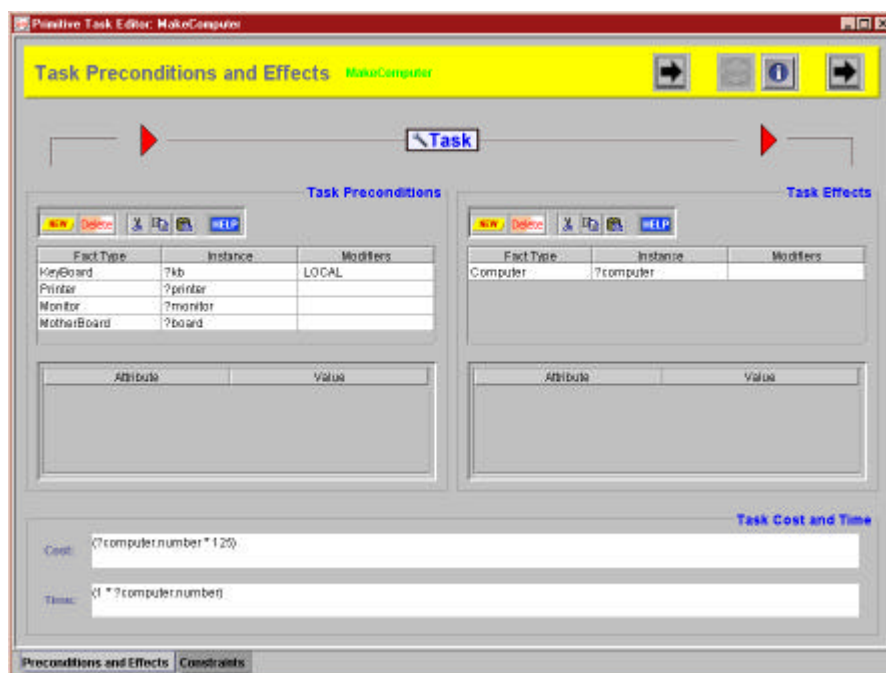


Figura 11.7: Interfície gràfica per definir tasques de tipus *primitive*. Es poden observar les zones per les precondicions i els efectes. En aquest cas per a poder ensamblar un ordinador cal tenir prèviament tots els elements que el componen: teclat, impressora, placa base i monitor.

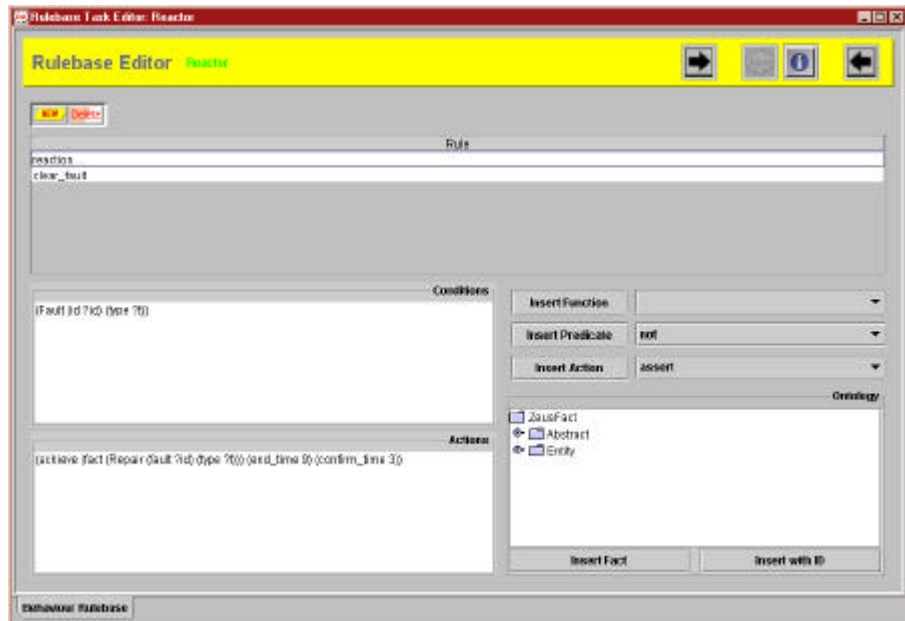


Figura 11.8: Editor de tasques tipus *rulebase*. Cada tasca pot contenir més d'una regla. Cada regla està formada per *condicions* que s'han de complir per executar la regla, i *accions* que es duen a terme si es donen les condicions.

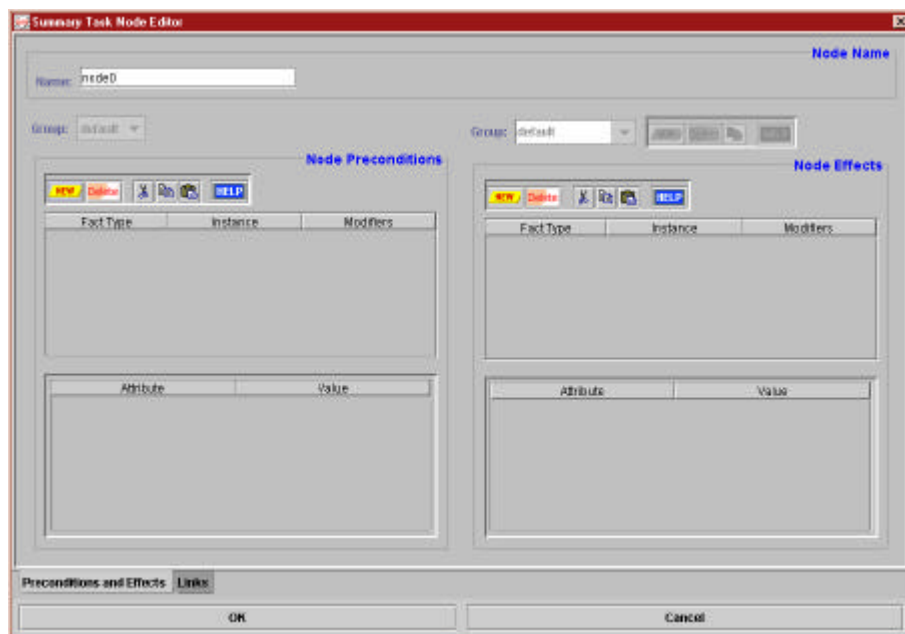


Figura 11.9: Editor de les tasques *summary*. L'estructura es semblant a la d'una regla, on es tenen unes precondicions i unes accions, però aquestes accions formen part d'algun pla i la tasca es veu com un node dins una seqüència de tasques.

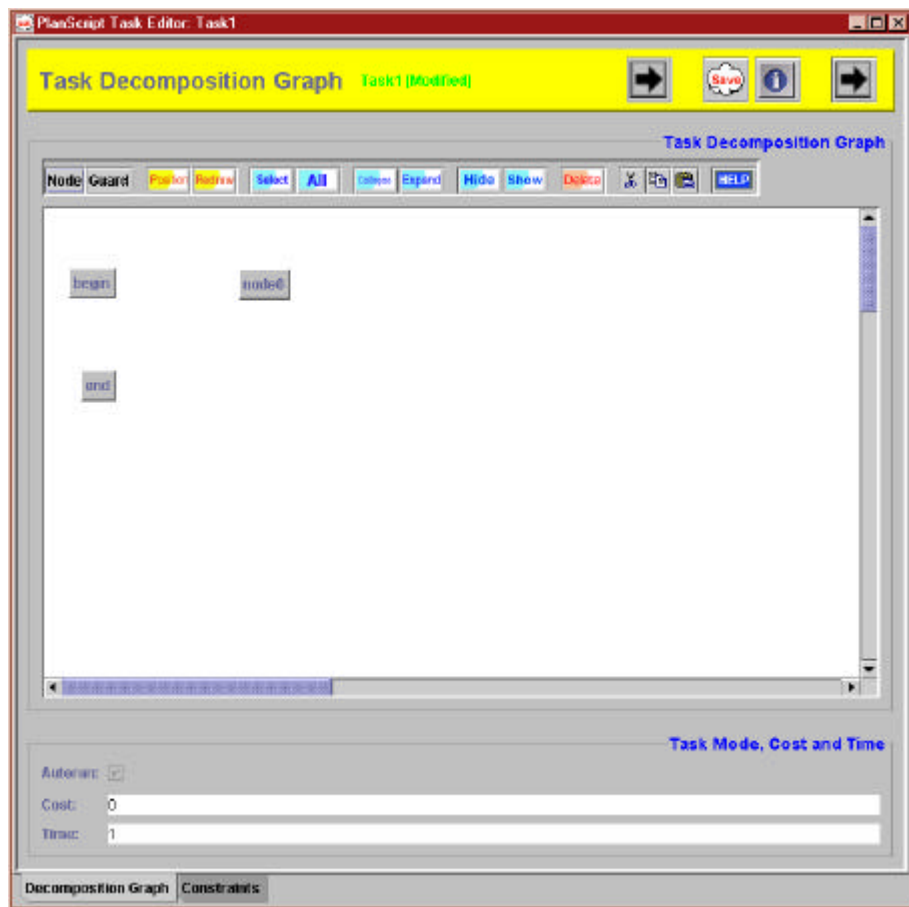


Figura 11.10: Editor d'una tasca *planscript*. Tot i que no es dóna molta informació sobre el funcionament, els *planscript* formen una seqüència d'accions que es poden dur a terme per aconseguir algun objectiu determinat.

ORGANITZACIÓ DE L'AGENT

En l'organització de l'agent, els agents són informats del seu context social en termes de les seves relacions amb els altres agents i de les habilitats que aquests posseeixen.

Les relacions que poden definir-se són quatre:

- 1) *Peer*: és la relació dels agents per defecte sense avançar cap tipus d'interacció amb altres agents.
- 2) *Superior*: Estableix una relació de superioritat d'un agent cap a un altre. Aquest agent pot donar ordres i els agents inferiors han d'obeir.
- 3) *Subordinate*: És la relació oposada a l'anterior. L'agent subordinat ha d'obeir les ordres que li arribin del seu agent superior.
- 4) *Co-worker*: És una relació semblant a la *peer*. En cas de requerir-se algun tipus de recurs, un agent demanarà ajuda a aquests agents abans de la resta.

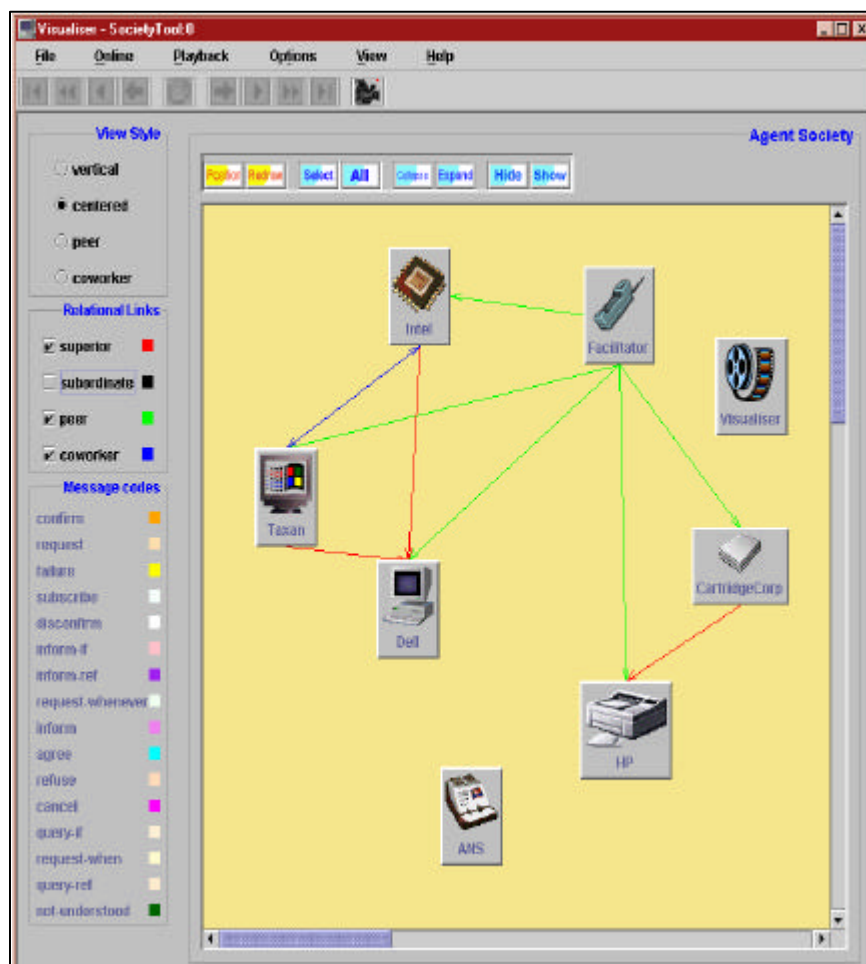


Figura 11.11: Organització en una comunitat d'agents. En aquest cas, es saben a priori diferents relacions entre els agents i poden definir-se.

En una de les utilitats de Zeus Visualiser (*SocietyTool*) es pot observar quina distribució tenen.

COORDINACIÓ DELS AGENTS

En la coordinació dels agents cada agent és equipat amb les habilitats socials i protocols per tal d'interaccionar amb la seva parella. Els protocols poden ser modificats amb estratègies que influenciaran en el comportament dels agents quan actuen.

Però abans de dur a terme aquest estadi, l'usuari ha de tenir una idea clara de com els agents interactuaran per tal de realitzar els seus respectius papers. Totes les interaccions de l'agent ZEUS són variacions del protocol *Contract Net* (v. Apartat 3.4.2).

El *Contract Net* suposa que es té un o més *Initiators* que emeten molts propòsits (missatge o performative de tipus *CFP*), i un o més *Respondent* que responen.

Els principals aspectes de qualsevol interacció d'agents són els protocols de coordinació i les estratègies de negociació.

El protocol de coordinació és un model de conversa d'agents que descriu quan cada part ha de comunicar-se, quins missatges seran intercanviats i l'efecte de rebre missatges particulars. El protocol de coordinació per defecte és descrit des d'ambdues perspectives, és a dir, des de qui inicia la negociació i des de qui respon, com es mostra a la Figura 11.12.

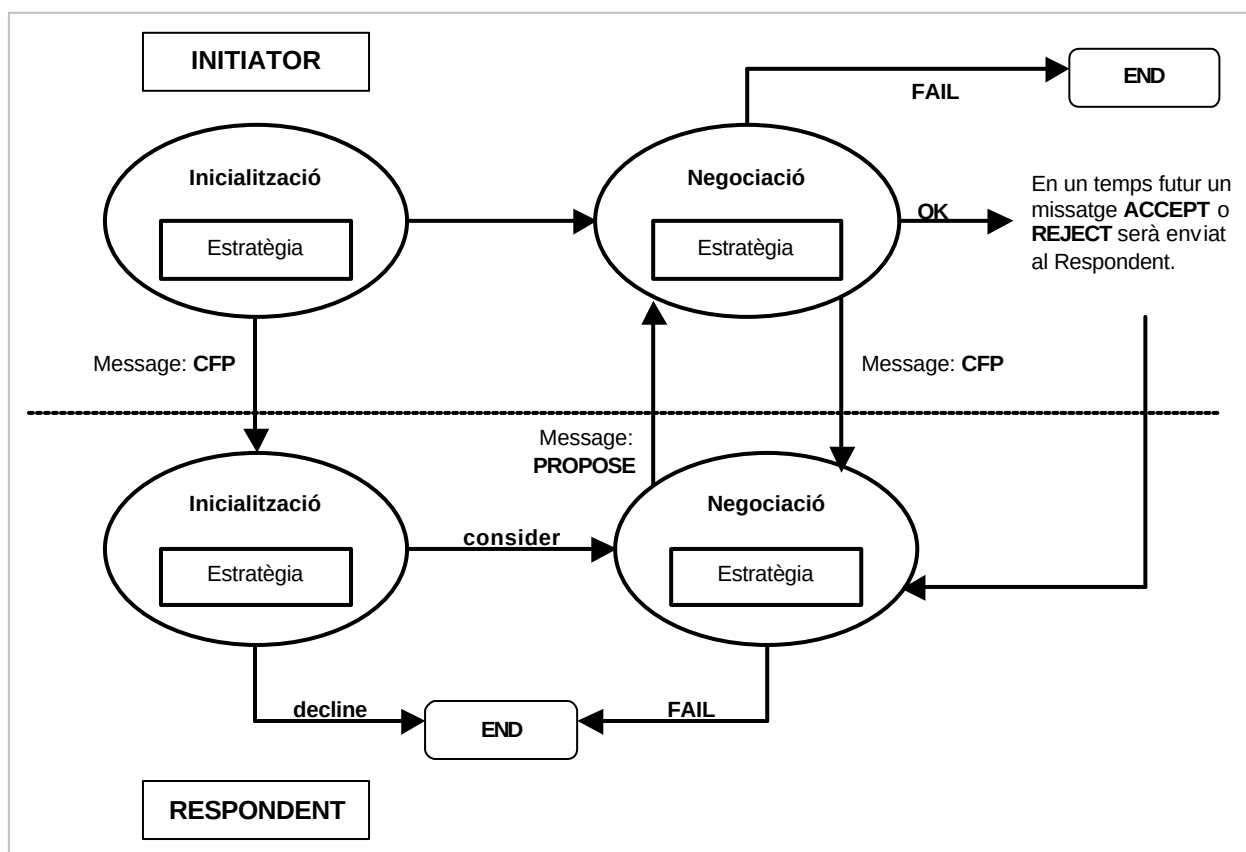


Figura 11.12: Esquema del *ContractNet protocol* implementat per Zeus.

Si un agent ha d'utilitzar una estratègia de coordinació l'*Agent Generator* té una interfície dedicada a tal efecte (v. Figura 11.13). Es té l'opció de triar diferents tipus d'estratègies per a dur a terme la negociació entre els agents. Les estratègies simulen les interaccions entre un comprador i un venedor, on el venedor comença demanant un preu molt alt i el comprador ofereix un preu molt baix, llavors la transacció es va succeint fins que es posen d'acord o no. Les transaccions es fan en termes de cost, temps i recursos obtinguts/oferts. Les diferents estratègies varien la forma en què ambdues parts van variant l'oferta/demanda.

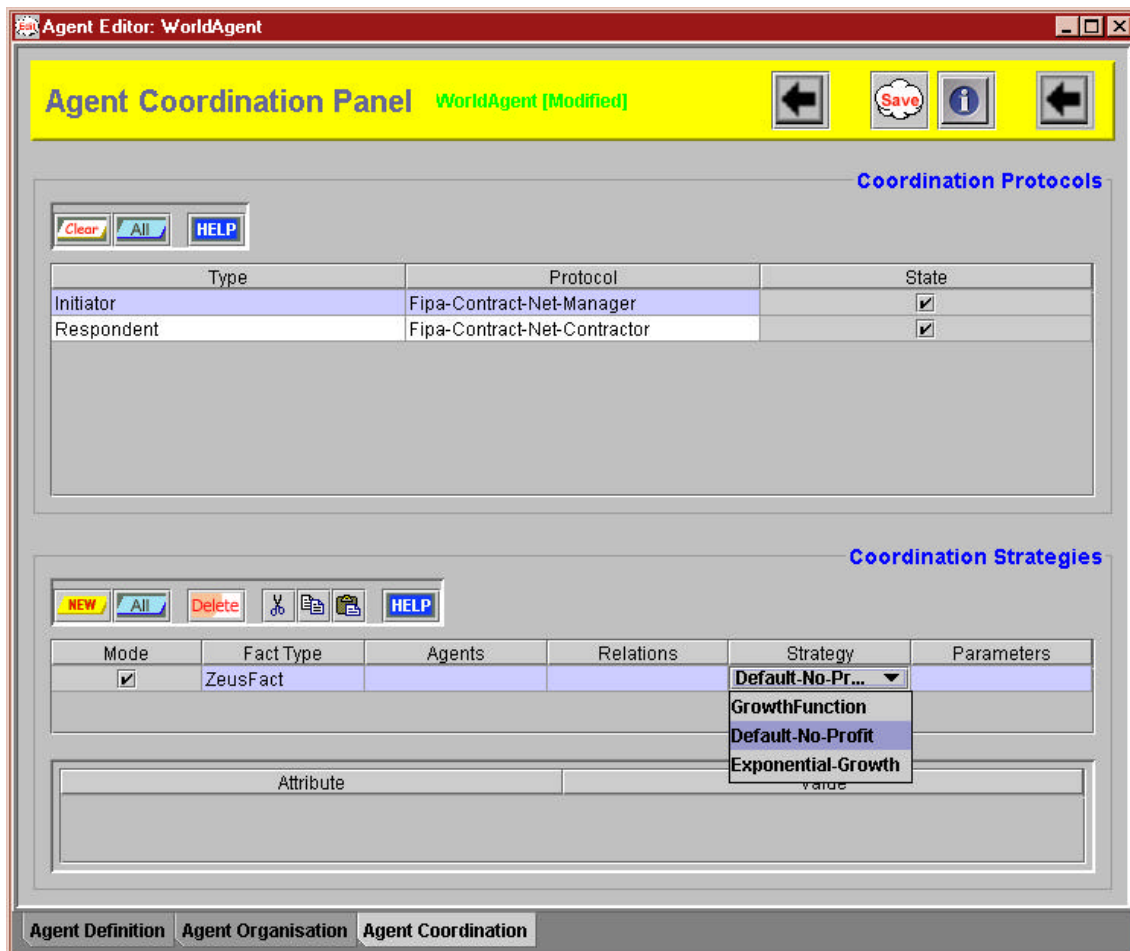


Figura 11.13: Panell on es defineix l'estratègia de coordinació que durà a terme l'agent tant per l'iniciador com per qui respon. Per defecte es tenen tres tipus. Es poden definir altres.

Veiem una conversa entre dos agents *SupplyBot* i *OrchadBot*, on el primer intenta comprar una caixa de pomes i el segon en ven una. Aquesta conversa s'ha extret d'un exemple que proporciona Zeus on tres agents simulen un mercat virtual de compra-venda de diferents productes.

SupplyBot is awaiting your instructions...

Attempting to buy: apple
 My preferences: price= 100, within 3 time-grains
 Conversation started...
 [cfp] to OrchardBot >> apple @ 0.000
 [propose] from OrchardBot >> apple @ 125.000
 [cfp] to OrchardBot >> apple @ 74.200
 [propose] from OrchardBot >> apple @ 122.766
 [cfp] to OrchardBot >> apple @ 78.400
 [propose] from OrchardBot >> apple @ 120.532
 [cfp] to OrchardBot >> apple @ 82.601
 [propose] from OrchardBot >> apple @ 118.298
 [cfp] to OrchardBot >> apple @ 86.801
 [propose] from OrchardBot >> apple @ 116.065
 [cfp] to OrchardBot >> apple @ 91.002
 [propose] from OrchardBot >> apple @ 113.831
 [cfp] to OrchardBot >> apple @ 95.202
 [propose] from OrchardBot >> apple @ 111.597
 [cfp] to OrchardBot >> apple @ 99.403
 [propose] from OrchardBot >> apple @ 109.364
 [cfp] to OrchardBot >> apple @ 103.603
 [propose] from OrchardBot >> apple @ 107.130
 [accept-proposal] to OrchardBot >> apple @ 107.130
 [result] from OrchardBot
 [invoice] from OrchardBot
 [payment] to OrchardBot

OrchardBot is awaiting your instructions...

Attempting to sell: apple
 My preferences: price= 100, within 3 time-grains
 Conversation started...
 [cfp] from SupplyBot >> apple @ 0.000
 [propose] to SupplyBot >> apple @ 125.000
 [cfp] from SupplyBot >> apple @ 74.200
 [propose] to SupplyBot >> apple @ 122.766
 [cfp] from SupplyBot >> apple @ 78.400
 [propose] to SupplyBot >> apple @ 120.532
 [cfp] from SupplyBot >> apple @ 82.601
 [propose] to SupplyBot >> apple @ 118.298
 [cfp] from SupplyBot >> apple @ 86.801
 [propose] to SupplyBot >> apple @ 116.065
 [cfp] from SupplyBot >> apple @ 91.002
 [propose] to SupplyBot >> apple @ 113.831
 [cfp] from SupplyBot >> apple @ 95.202
 [propose] to SupplyBot >> apple @ 111.597
 [cfp] from SupplyBot >> apple @ 99.403
 [propose] to SupplyBot >> apple @ 109.364
 [cfp] from SupplyBot >> apple @ 103.603
 [propose] to SupplyBot >> apple @ 107.130
 [accept-proposal] from SupplyBot >> apple @ 107.130
 [result] to SupplyBot
 [invoice] to SupplyBot
 [payment] from SupplyBot

El primer pas és la inicialització de la conversa. Una vegada acceptada per les dues parts, es comença l'oferta/demanda. *SupplyBot* comença demanant 125 (s'ha configurat que la negociació es comença amb el preu base més un 25%), i al final accepta 107.130 del comprador. *OrchadBot* inicialment ofereix 74.200 (igual que *SupplyBot* s'ha configurat un marge inicial d'un 25%).

11.2.2.3. Configuració de les utilitats dels agents

Una vegada les tasques han estat dissenyades queda per dissenyar la infraestructura coneguda per utilitats dels agents.

Abans de realitzar aquesta fase cal tenir clars diversos punts:

- La resolució del sistema. L'ANS té un rellotge intern que pot ser configurat amb un divisor d'un minut.
- Si es farà servir o no un facilitador, i en cas que calgui fer-lo servir, cal configurar els seus paràmetres. El facilitador té la missió de fer de mitjancer en transaccions de negociació.
- Si es faran servir recursos externs. Els recursos poden ser de tots tipus: bases de dades i programes externs.

Les utilitats dels agents es configuren al panell *utility panel* de *ZEUS Agent Generator*. En configuren tres agents que sempre són presents en el sistema: l'ANS (*Agent Name Server*), el *Facilitator* (agent mitjancer), i el *Visualiser* (agent de l'entorn visual de Zeus). A més d'aquest tres agents, es configura el *database proxies*.

AGENT NAME SERVER

Cada comunitat d'agents ha de tenir un agent ANS. Aquest agent manté una agenda de tots els agents coneguts, permetent que els agents puguin transmetre missatges entre ells i la seva localització.

Els paràmetres de configuració d'un ANS es poden veure a la Figura 11.14.

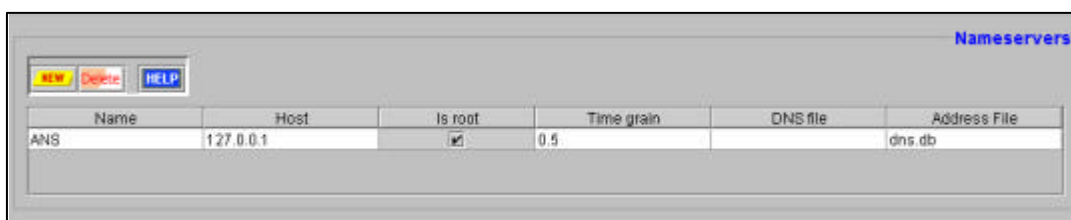


Figura 11.14: Panell de configuració dels ANS.

Si un ANS és root, pot configurar el rellotge intern que governa la transmissió de missatges dins la comunitat d'agents.

AGENT FACILITADOR

L'agent facilitador és un agent especialitzat en realitzar de mitjancer entre un agent que demana un recurs i la comunitat d'agents que el proporciona. Té coneixement de les relacions entre els agents i llurs habilitats.

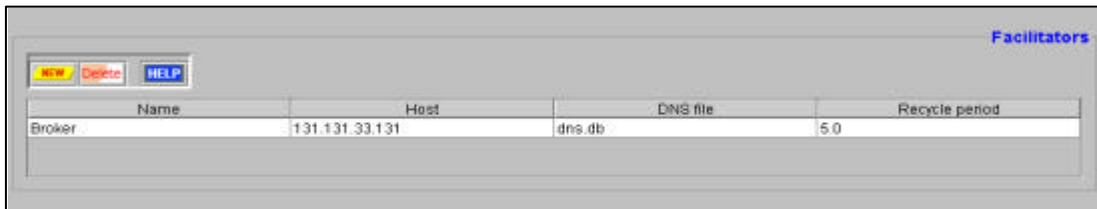


Figura 11.15: Interfície de configuració del facilitador. S'ha de donar l'adreça IP de la màquina. El *recycled period* és un valor expressat en minuts.

El facilitador periòdicament (segons el *recycle period*) recopila informació de tots els agents del sistema enviant un missatge qüestionant-los sobre les habilitats que poden dur a terme en aquell moment, així el facilitador manté la informació al dia. Si el valor de *recycle period* és 0, el facilitador actua reactivament quan se li demana.

AGENT VISUALITZADOR

Aquest agent no és imprescindible en el funcionament d'una aplicació. Aquest agent entra en joc si nosaltres volem veure com funciona el sistema de manera gràfica. Aquest agent engega una interfície que és *ZEUS visualiser* on l'usuari té diferents opcions molt útils.

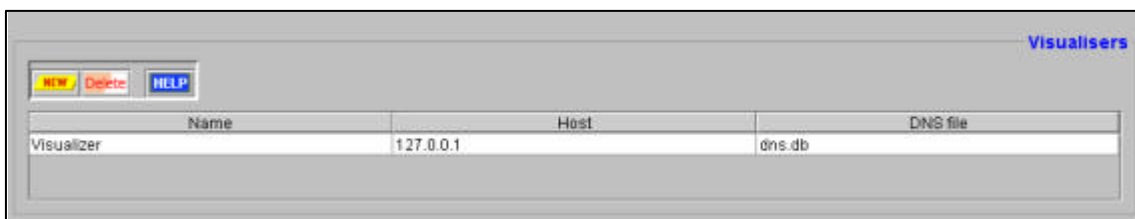


Figura 11.16: Interfície per configurar el visualitzador. La configuració només demana la localització (adreça IP) de l'agent.

DATABASE PROXY

Un agent de tipus database proxy proporciona unes funcions afegides al sistema. Pot servir com interfície d'una base de dades, o traduir informació que ha de ser manegada en codi ASCII.

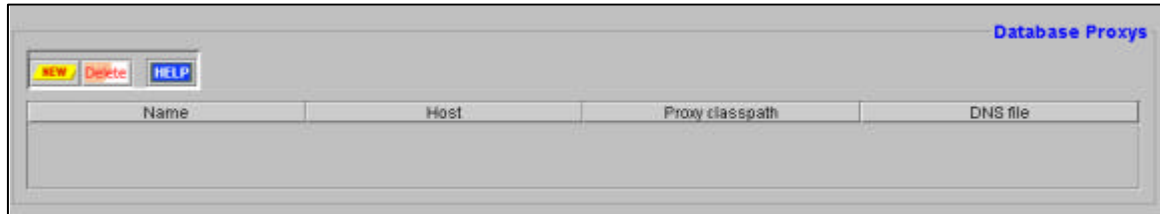


Figura 11.17: Configuració dels *database proxys*. Cada entrada ha de configurar la seva localització física (adreça IP) i lògica (path o camí dins el sistema de fitxers).

11.2.2.4. Configuració de les tasques de l'agent

Una vegada s'ha configurat l'infraestructura dels agents del sistema (*ANS*, *Facilitator* i *Visualiser*), cal fer un procés semblant amb els agents.

En aquesta fase es configuren característiques dels agents. Si veiem la Figura 11.18, cadascun dels agents té diferents paràmetres.

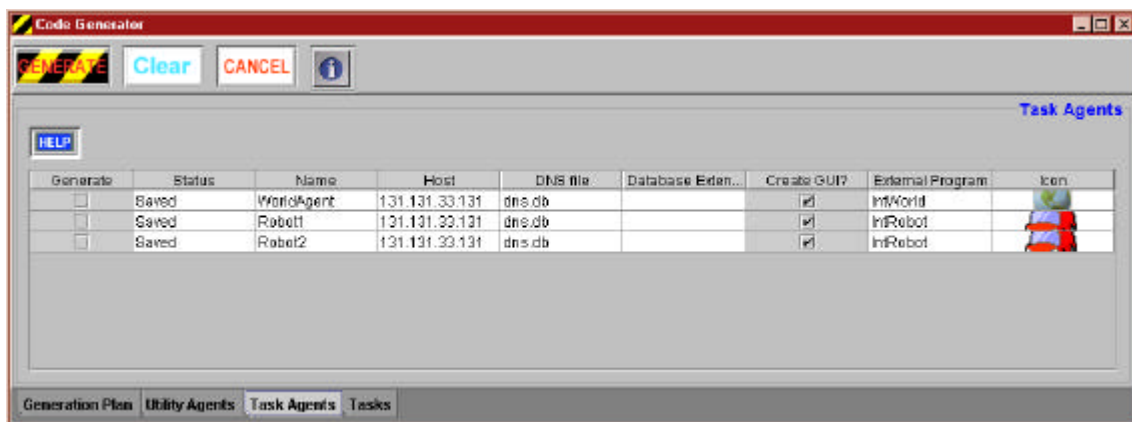


Figura 11.18: Panell de configuració de les tasques dels agents. El panell es mostra en clicar en la icona de *Generate Code* de *ZEUS Agent Generator*.

Per cada agent cal determinar diferents paràmetres/recursos:

- *Generate*: és una opció per forçar la generació del codi de l'agent. Per entendre millor aquesta opció, en l'apartat següent s'explica la utilització.
- *Host*: cal determinar l'adreça IP de l'agent.
- *DNS file*: fitxer de text on es guarda informació de la configuració.
- *Database extern*: Especificar el package que es farà servir (si cal).
- *Create GUI*: Especifiquem si es vol crear una interfície per monitoritzar el funcionament de l'agent.
- *External program*: Especifiquem el nom del programa (si cal) que farà servir l'agent.
- *Icon*: Per visualitzar l'agent, aquest es simbolitza en una icona.

11.2.2.5. Implementació de l'agent

La última fase abans d'executar l'aplicació és generar el codi font de tots els fitxers involucrats.

Aquest procés està pensat de manera que l'usuari de Zeus clicant en una icona, genera tots els codis que cal. És a dir, si un agent s'ha modificat o reconfigurat en alguna cosa, Zeus en té coneixement i li dóna l'*status* de *modified*; llavors quan es vol generar el codi font, es genera. Si no ha estat modificat des de l'última generació no s'interpreta. També es pot forçar la generació de codi de tots els agents i tasques que l'usuari desitji, activant l'opció *generate* de la configuració de tasques dels agents.

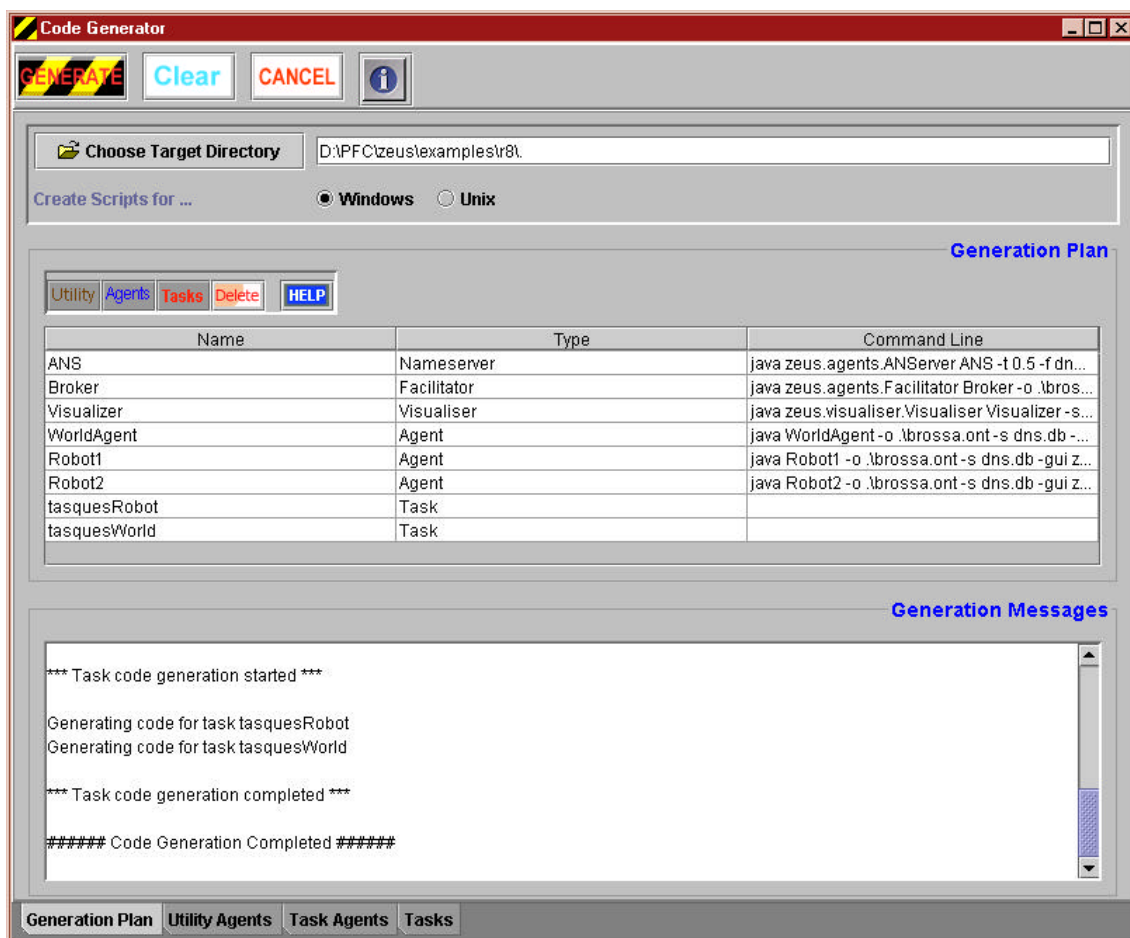


Figura 11.19: Interfície gràfica de *Code Generator* amb els resultats de la generació del codi font dels agents, tasques i fitxers per executar l'aplicació.

11.3. Arquitectura

L'arquitectura de Zeus es pot veure en conjunt o en un agent en particular. És a dir, es té un conjunt de components escrits en Java que proporcionen diferents tipus de funcions al voltant de l'agent; i més concretament, un agent està format per diferents elements.

11.3.1. Entorn de desenvolupament

L'entorn de desenvolupament està format per una llibreria de components d'agents (*Agent Component Library*), software de construcció (*Agent Building Software*) i utilitats dels agents (*Utility Agents*).

11.3.1.1. Llibreria de components d'un agent (*Agent Component Library*)

La llibreria de components dels agents és un conjunt de classes que permeten construir agents com blocs independents (amb llurs funcions) i donar-los prestacions per tal de col·laborar/parlar amb altres agents.

Aquesta llibreria pot veure's en quatre nivells: *comunicació* (transmissió), *ontologia* (representació del coneixement), *coordinació* (interacció amb altres agents) i *raonament* (plans, tasques i eines per tal d'aconseguir objectius).

- *Comunicació*: La comunicació s'estableix entre els diferents agents del sistema per diferents raons. Cal transmetre missatges en un format que ambdós interlocutors entenguin.

En aquest cas s'ha escollit el protocol de transmissió de dades TCP/IP que treballa a través de sockets, i el format dels missatges basat en KQML (*Knowledge Query and Manipulation Language*).

- *Ontologia*: L'ontologia és un entorn que proporciona a l'agent la forma de guardar/manipular el seu coneixement. El coneixement es guarda en estructures de dades que són identificades per noms (igual que variables de dades) que contenen diferents camps de diferents tipus. L'ontologia és compartida per tots els agents que formen part de la comunitat, però cadascun d'ells la instància/executa localment. Per exemple, es pot tenir un tipus de l'ontologia anomenat posició que conté les coordenades d'un element, però son cadascun dels agents que el facin servir, què l'inicialitzen amb els valors que tria.

La transmissió de dades es fa basant-se amb els elements de l'ontologia, i es transmeten missatges que contenen aquests elements.

- *Coordinació*: Zeus proporciona eines que intervenen en aquest nivell social. Es proporcionen estratègies (algoritmes) per tal de dur a terme tasques coordinades (per exemple es facilita l'ús del protocol de *Contract Net* de comerç electrònic, i la possibilitat de definir noves estratègies) o de dur a terme plans predissenyats, o de programar tasques que tinguin a veure amb el coneixement de l'agent (regles de comportament), etc. Zeus permet definir relacions entre agents. Aquestes relacions són de mestre-esclau (un agent està sotmès a un altre), *peer* (parella) o *coworker* (col·laborador), amb diferent tractament de les estratègies segons el cas.
- *Raonament*: Els agents són capaços de realitzar funcions per tal d'aconseguir algun fet determinat. El raonament s'entén com la forma d'actuar davant d'aquestes fites. En l'entorn Zeus podem triar entre diferents tipus de tasques que un agent pot realitzar. Segons l'aplicació i les funcions dels agents caldrà veure quin tipus implementar. Com s'ha comentat abans, es tenen quatre tipus bàsics de tasques: *primitive*, *rulebase*, *planscript* i *summary*.

11.3.1.2. Software de construcció (Agent Building Software)

Aquest entorn de construcció d'agents té la finalitat de permetre construir aplicacions específiques de sistemes multiagent proporcionant un entorn el més genèric possible. L'entorn està format per dos elements: una metodologia de desenvolupament basada en les prestacions de l'ontologia i les funcions que poden dissenyar-se, i un entorn visual que permet desenvolupar aquestes aplicacions.

La metodologia és el procés que s'ha de seguir per dissenyar una aplicació multiagent. En l'apartat anterior es dóna una descripció detallada del mètode que es pot seguir per destriar tots els elements que ens faran falta per la implementació sobre Zeus.

L'entorn visual per implementar l'aplicació (v. Figura 11.20) està molt treballat i consta de nombroses pantalles i opcions per permetre molts tipus d'aplicacions. Està pensat per un ampli ventall de dominis i realitzat de forma molt intuïtiva i fàcil d'utilitzar.

Aquesta eina de desenvolupament conté diferents elements.

Les opcions del projecte permeten de generar els codis font dels agents i les tasques; a més, permet de veure els agents i les seves relacions gràficament.

Es té un editor per desenvolupar una ontologia específica per cada problema. L'editor permet de crear una base de coneixement amb fets (aplicables a regles de comportament) i variables.

L'editor d'agents permet definir els agents. S'especifiquen les tasques que duren a terme cadascun dels agents, els recursos que tenen inicialment (instàncies de les variables de l'ontologia), la relació que tenen els agents entre ells, si es té o no relació amb bases de dades, i l'estratègia que ha de seguir en cas de conversar amb altres agents per aconseguir algun fet.

L'editor de tasques permet definir diferents tipus de tasques que poden dur a terme els agents.



Figura 11.20: Construcció dels agents. Interfície de generació del sistema: Zeus Agent Generator.

11.3.1.3. Utilitats dels agents (ZEUS Utility Agents)

En la metodologia de desenvolupament s'ha vist que les utilitats que proporciona el sistema són els agents *Facilitator*, *Visualiser* i *Name Server*.

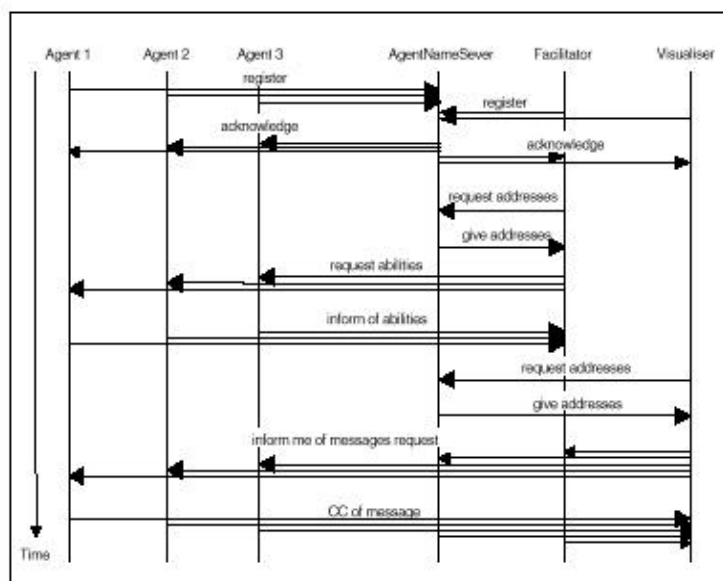


Figura 11.21: Procés d'inicialització dels agents que engega el sistema per defecte. Els agents de l'aplicació comencen a treballar si tot aquest procés ha finalitzat amb èxit.

Cada agent realitza una funció determinada dins el sistema Zeus i la seva feina es fa en segon terme (background).

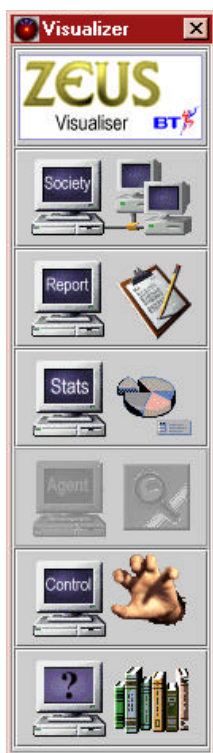


Figura 11.22: Opcions de la interfície de *Visualiser*.

L'excepció d'aquesta feina és l'agent *visualiser*. Aquest agent engega una interfície molt útil per tal d'ajudar a l'usuari a monitoritzar el funcionament de l'aplicació.

Té diverses opcions. Es pot observar el funcionament de la comunitat d'agents (*society view*), generar estadístiques de diferents tipus, i observar les tasques que es van generant. És molt útil.

Tot i que no forma part de l'agent *visualiser*, es pot definir que cada agent generi una interfície gràfica que mostra diversos elements referents al funcionament dels agents. Es té informació dels

missatges que ha generat i rebut, les regles que s'han anat activant, les tasques que s'han executat, els recursos que té, etc. És recomanable generar-la sempre per observar les evolucions de l'agent.

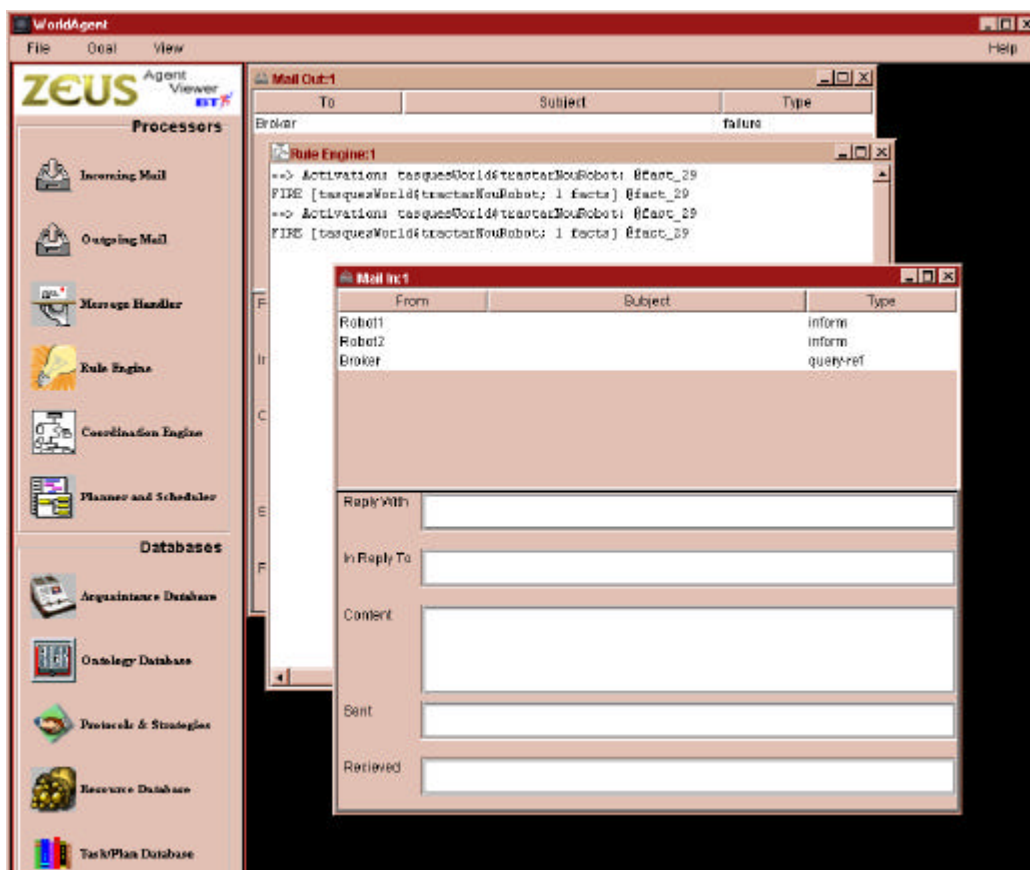


Figura 11.23: Interfície gràfica d'un agent. En aquest cas es mostren els missatges que ha rebut l'agent, i les regles que s'han activat fins al moment.

Una de les eines que proporciona la interfície de *visualiser* és generar estadístiques de l'aplicació. No és una eina imprescindible, però és d'agrair si es vol estudiar el rendiment d'una aplicació.

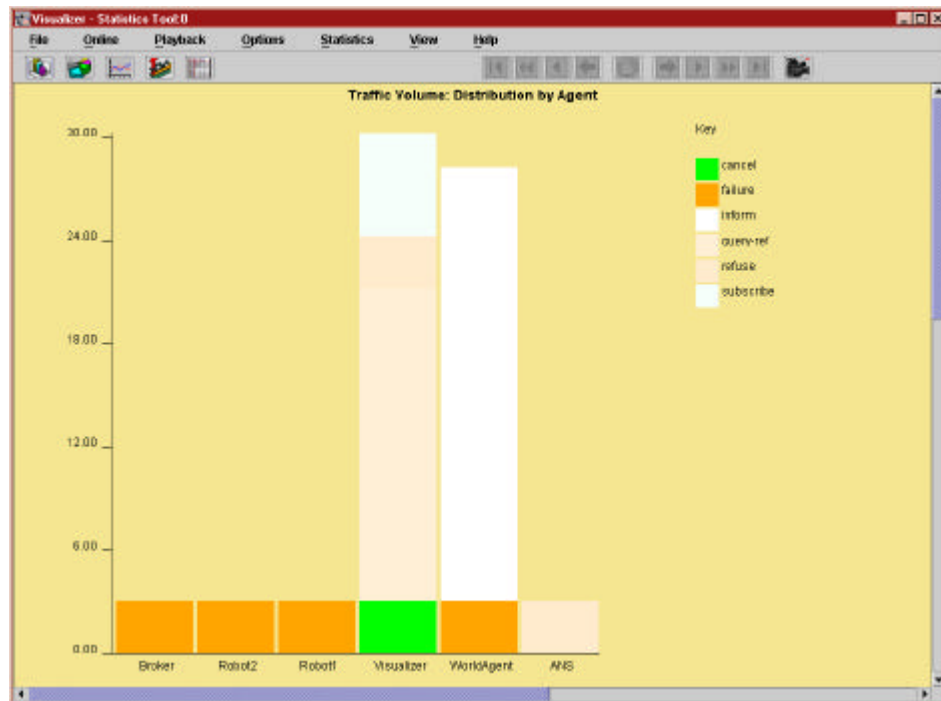


Figura 11.24: Interfície gràfica que mostra una estadística de missatges enviats per cada agent.

11.3.2. Descripció d'un agent

Les eines que proporciona Zeus permeten dissenyar els agents de forma independent. Tot i la transparència que proporciona en la implementació interna de l'agent, aquest té una arquitectura col·laborativa que es pot veure i analitzar per treure conclusions de com funciona l'agent.

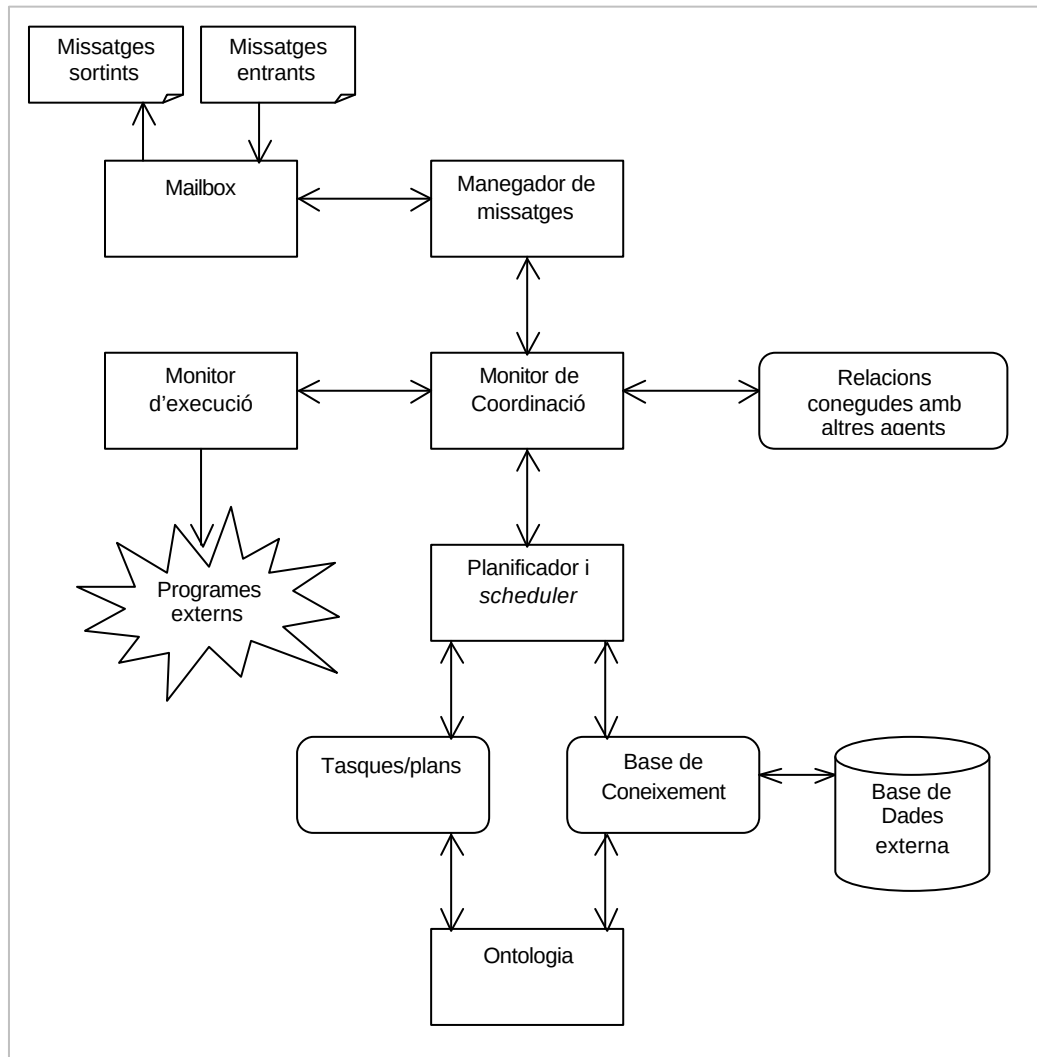


Figura 11.25: Arquitectura d'un agent Zeus.

Elements de l'agent:

- *Mailbox*: manega les comunicacions entre l'agent i la resta.
- *MessageHandler* o Manegador de Missatges: processa els missatges entrants del *Mailbox*, i transmet el contingut als components de l'agent a qui correspon a través del Monitor de Coordinació.
- El *Monitor de Coordinació* és el cor de l'agent. Pren decisions tenint en compte els objectius de l'agent i la situació actual del sistema. És responsable de realitzar la coordinació amb altres agents utilitzant les estratègies que l'agent implementi (per exemple el *Contract Net Protocol*).
- *Relacions conegudes amb altres agents*. Es descriuen les relacions que l'agent té amb la resta de la comunitat. Aquesta informació és utilitzada pel *Monitor de Coordinació* a l'hora de prendre decisions.
- *Planificador i scheduler*. Monitoritza diferents tasques implementades pels agents.
- *Base de Coneixement* o *Resource Database*. Conté una llista dels recursos (*facts*) que té/posseix l'agent. També es permet interactuar amb BBDD pròpies de l'agent o compartides.
- *Task/Plan Database*. Conté la descripció dels plans i accions que pot realitzar l'agent. Les accions poden fer referència a l'*ontologia* de l'agent.
- *Ontologia*. Manté la definició lògica de cada element de la *Base de Coneixement* (*facts*).
- *Execution Monitor* o *Monitor d'execució*: manté un rellotge intern, que inicia, acaba o monitoritza totes les accions que s'estan executant en el sistema. També monitoritza l'execució dels programes externs que s'executin.

11.4. Desenvolupament d'una aplicació

11.4.1. Descripció de l'aplicació

L'aplicació és la del *món de la brossa*.

En el nostre sistema es tenen dos tipus d'agents: el *món* i els *robots*. El món monitoritza l'execució del sistema i va generant brosses en una posició aleatòria. Els robots, que estan esperant, quan detecten aquesta nova brossa, hi van des d'on siguin. En aquest cas s'ha implementat una petita estructura de dades a l'agent *món* per tal d'assignar feina al robot que és més proper de la brossa, així evitem que hi vagin tots els robots alhora.

11.4.1.1. Inici del sistema

L'entorn Zeus dona moltes facilitats per iniciar el funcionament d'una aplicació. En la utilitat *AgentGenerator* es té l'opció de *Code Generator*. Aquesta utilitat té la funció de configurar diferents paràmetres dels agents (per exemple, l'adreça IP de cadascun, especificar si un agent utilitzarà o no programes externs, o si es crearà una interfície per monitoritzar les accions/resultats de cada agent, etc.), verificar que la definició dels agents sigui correcta i no hagi inconsistències (com per exemple, si els agents tenen recursos inicials, es verifica que tots els paràmetres estiguin inicialitzats), i finalment si tot és correcte, es generen els fitxers de sortida del sistema.

Cada agent del sistema es converteix en un fitxer Java. El nom d'aquests fitxers es correspon amb el nom que s'ha designat a la interfície *AgentGenerator*. Abans d'executar l'aplicació cal compilar cadascun d'aquests fitxers.

Les tasques també són traduïdes en fitxers. En el nostre cas, les tasques que es tenen són del tipus *rulebase*. Aquestes regles de comportament es tradueixen en fitxers de text que són interpretats pel sistema. El nom dels fitxers es correspon amb el nom que s'ha donat en definir les tasques. L'extensió d'aquests fitxers és *clp*. La forma d'aquestes tasques emula les definicions de *clips*.

Les regles del tipus *primitive* es converteixen en fitxers Java i cal compilar-los abans de començar.

A més d'aquests fitxers es generen tres fitxers per lots (*run1.bat*, *run2.bat* i *run3.bat*) que permeten iniciar l'execució de l'aplicació fàcilment.

Quan donem per finalitzada l'etapa de definició de tot l'entorn, s'han de generar els fitxers que seran la base de l'aplicació. Sempre cal observar els missatges que es donen a l'àrea de text de la interfície *Code Generator* per veure possibles errades. Es dona informació del tipus d'errada i l'agent on es troba.

En la nostra aplicació, si tot és correcte podem observar els missatges que es donen i veure els fitxers que després haurem de compilar (v.Figura 11.26).

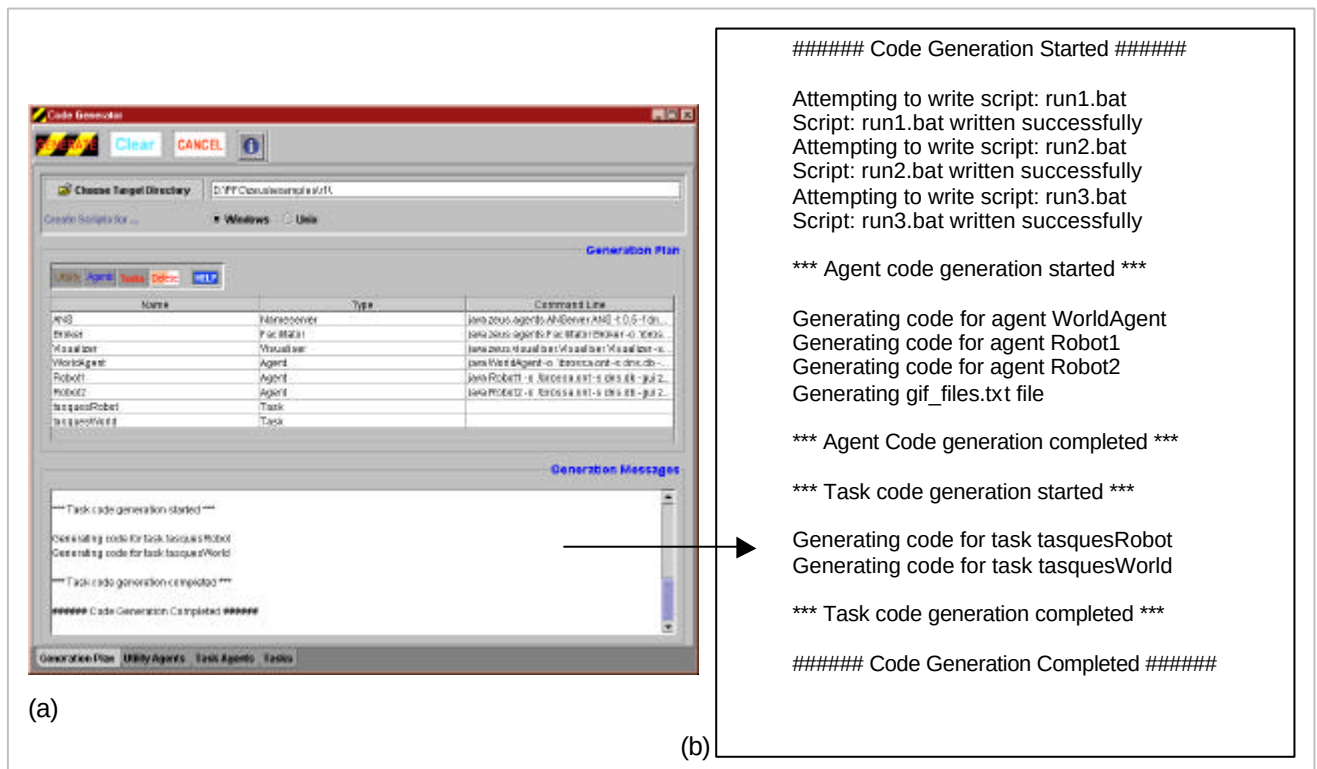


Figura 11.26: a) Interfície gràfica *Code Generator* d'*AgentGenerator* que permet generar els fitxers de sortida dels diferents elements del sistema (agents, tasques, ..);
b) Missatges de sortida de *Code Generator* informant dels passos que ha realitzat i el seu resultat. En aquest cas tot es correcte i ja es podrien compilar per iniciar l'execució.

Així en aquest cas s'han de compilar els fitxers *WorldAgent.java*, *Robot1.java* i *Robot2.java*.

No sempre cal compilar-los tots. El sistema permet al programador dir quin/s fitxers generar en cada moment, però sempre genera els fitxers dels agents o tasques que s'han modificat des l'última generació de codi (això està molt bé, no cal preocupar-nos).

Donat que la compilació de fitxers Java pot ser lenta, un consell és generar un fitxer per lots i anar-hi definint els fitxers que volem compilar en cada moment.

A més dels fitxers de sortida del sistema generats automàticament, es poden utilitzar fitxers externs. Aquests fitxers s'han de compilar abans d'iniciar l'aplicació i han d'estar lliures d'errors. Aquesta condició és necessària però no suficient. En l'experiència de programar aquesta aplicació m'he trobat en diferents situacions que poden portar molts maldecaps. Encara que el fitxer extern estigui lliure d'errors, pot ser que no pugui ser executat correctament per l'agent. La detecció d'aquest fet és senzilla doncs l'agent en qüestió no s'inicia i es tanca la finestra de Java que ha iniciat. La solució ja és més complicada, doncs això pot ser degut a diferents causes. Pot ser degut a la mala utilització dels recursos de Zeus, com per exemple donar un nom incorrecte a un fet (*fact*) que utilitzem, a la mala inicialització d'alguna eina gràfica (*frame*, *panel*, o d'altres elements gràfics), o d'altres causes. Abans d'avortar l'execució, en la finestra de Java es mostra el missatge de l'excepció i on s'ha produït. Cal capturar el missatge abans que es tanqui la finestra. Depenent de l'ordinador on s'executi l'aplicació, aquest

fet pot ser ràpid i no es pot llegir, llavors una possible solució és capturar la pantalla (en Windows fent servir les tecles *Alt+ImprPant*, o en Linux, fent servir l'aplicació *xv*).

En el nostre cas es tenen dos fitxers externs: *IntWorld.java* i *IntRobot.java*. Aquests dos tipus de fitxers tenen la funció de mostrar les evolucions del sistema, i en el cas del món, l'usuari dóna l'ordre de generar una nova brossa. No és una simulació de durada i periodicitat fixada. Es poden anar veient les evolucions i com es treballa. Són les interfícies gràfiques dels nostres agents.

Una vegada generats i compilats tots els fitxers del sistema es pot iniciar l'execució de l'aplicació.

Com s'ha vist Zeus genera tres fitxers *bat*. Són els fitxers per iniciar l'execució.

Primer s'executa *run1.bat*. Si tot va bé s'inicia l'execució en background de l'Agent Name Service de Zeus. En totes les aplicacions s'inicia. És l'agent encarregat de tenir la informació de la localització dels agents del sistema. Els agents quan han de realitzar alguna transmissió li demanen informació del receptor i aquest els hi dóna. És un ANS centralitzat.

Segon, s'executa *run2.bat*. Aquest fitxer inicia dos agents més imprescindibles d'una aplicació Zeus. Són el *facilitator* i el *visualizer*. El *facilitator* té la missió d'intervenir en transaccions entre agents. Una de les principals funcions que realitza Zeus és la negociació per algun fet concret (per exemple, una compravenda entre agents), llavors el *facilitator* proporciona la infraestructura necessària per dur a terme aquestes converses. El *visualizer* és un agent que s'utilitza per donar prestacions visuals a una aplicació Zeus. Una de les aplicacions que proporciona Zeus és una interfície gràfica on hom pot veure quins agents hi ha al sistema i veure com es transmeten missatges entre ells. Per realitzar aquesta funció de manera independent es té aquest agent. Si l'usuari no fa servir aquesta eina (*SocietyTool* d'*Agent Visualizer*) aquest agent no fa res, però si es fa servir, els agents s'hi subscriuen i aquest monitoritza tots els seus moviments.

Per últim s'executa *run3.bat*. Aquest fitxer inicia els agents propis de l'aplicació. En el nostre cas s'inicien els robots i el món, amb llurs interfícies gràfiques definides per nosaltres.

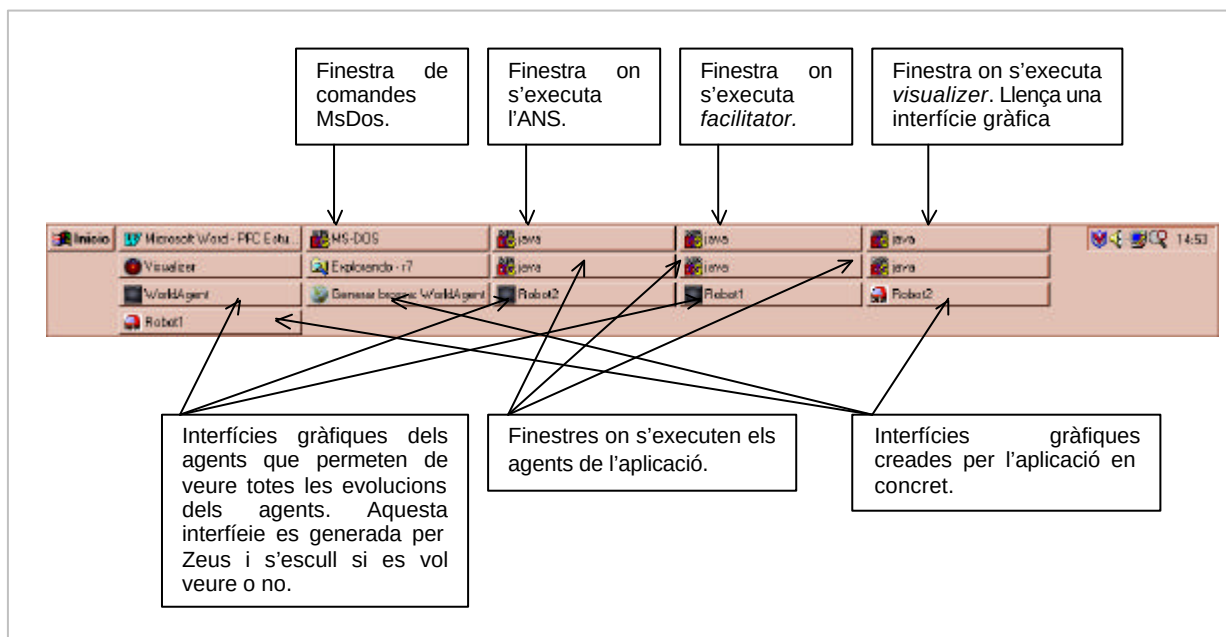


Figura 11.27: Barra d'eines de Windows95 amb l'aplicació inicialitzada correctament. Poden observar-se tots els elements que s'han creat. Donada la complexitat del sistema, pot requerir-se que la màquina on s'executi l'aplicació sigui potent, perquè podria resultar una execució lenta. De fet, aquest és un dels principals problemes que té aquest entorn de desenvolupament.

11.4.1.2. Creació dels agents

En la Figura 11.28 es pot veure com la inicialització del sistema engega els diferents agents del sistema. Abans d'iniciar el funcionament cal definir-ne la estructura/funcionalitat de cadascun.

La creació dels agents es fa a l'aplicació *AgentGenerator* (v.Figura 11.28) proporcionada per Zeus.



Els agents es defineixen a l'apartat d'opcions dels agents (*agent options*). Quan es creen els agents se'ls dona un nom i ja està. Després s'inicialitzen i se'ls donen les propietats/funcions que els caracteritzaran.

En el nostre cas es defineixen tres agents: WorldAgent, Robot1 i Robot2. En l'apartat següent explicarem com funcionen, però ara avancem que les tasques que realitzen els agents, tasquesRobot i tasquesWorld, són regles de comportament que ens permetran realitzar la comunicació entre els diferents agents.

A banda de la definició de les tasques que realitzen els agents, s'han de definir altres característiques.

Figura 11.28: Interfície gràfica d'AgentGenerator.

El disseny d'una solució per un problema no té perquè ser única. En el disseny dels agents s'ha triat una possible solució, que no té perquè ser la millor.

Veiem les característiques de cada agent per separat:

WorldAgent: Quan editem l'agent (*AgentEditor*), es poden definir els recursos inicials de l'agent, la relació que manté un agent amb la resta, i si s'ha de realitzar algun tipus de negociació, es pot definir la estratègia a seguir.

En el nostre cas, com la feina més important la fa la interfície que s'ha dissenyat, no cal assignar cap valor a cap element.

Robot: Els robots són tots iguals. En el disseny de l'aplicació s'ha procurat realitzar-la de manera que els robots fossin tant independents com fos possible. Així s'ha conseguit que puguin clonar-se els agents robots sense modificar res més.

Així, els robots no tenen coneixement a priori ni tampoc cap tipus de relació amb altres agents.

Aquesta solució triada no és la primera que es va pensar. Es va pensar en una relació client-servidor entre el món i els robots, de manera que poguéssin negociar qui anés a recollir una brossa. Més endavant es va veure que aquesta solució no era factible per les estratègies que té Zeus per a realitzar les negociacions basades en *contractNet*.

La estratègia que es fa servir per negociar es basa en converses dos a dos, és a dir, dos agents negocien per a assolir un fet, però no es pot realitzar una conversa entre més de dos. Això suposa que el nostre problema no es podia ampliar, doncs la solució que es podia trobar, només serviria per dos robots. A més de les converses basades en el *contractNet* es té una eina anomenada *achieve* per aconseguir un fet. Aquest fet també s'aconsegueix mantenint converses entre agents. Ara bé, ni la documentació ni un exemple que la fa servir, no aclareixen el seu funcionament/ús, així que es va desistir de la seva utilització.

A més de tot això, es va demanar consell a la persona responsable d'aquest projecte, Jaron Collins, i aquest va recomanar, entre altres alternatives pel nostre problema, la solució que s'ha implementat.

Això no vol dir que aquestes eines no siguin molt útils en altres dominis, sinó que per al nostre problema no és la millor solució.

Finalment, abans de passar a les tasques que realitzen els agents, i abans de donar per creats els agents cal donar-los una localització física, una icona, una interfície gràfica per monitoritzar-los i generar-ne el codi.

Zeus per a definir aquestes opcions/característiques proporciona una interfície molt bona: *Code Generator*.

La interfície *Code generator* consta de quatre panells: *Generation Plan*, *Utility Agents*, *Task Agents* i *Tasks*.

El primer genera els fitxers de sortida dels diferents agents del sistema i tasques; el segon, proporciona la interfície per definir les adreces IP i altres característiques dels agents imprescindibles del sistema (*ANS*, *Facilitator* i *Visualiser*); el tercer, és la interfície per caracteritzar els agents de l'aplicació (en el nostre cas es dóna la seva localització com a local de la màquina, com no s'han modificat no cal generar de nou el codi, es tria que hagi una interfície per monitoritzar el seu funcionament (*Create GUI?*), es dóna el nom del fitxer extern que utilitzaran i, finalment, s'assigna una icona per utilitzar en visualitzacions de la comunitat d'agents); el quart i últim panell, dóna informació de les tasques definides al sistema i es pot forçar la generació del seu codi (si es vol, encara que no s'hagi canviat el codi).

En la Figura 11.29 es poden veure els diferents panells i els valors dels paràmetres de *Code Generator*.

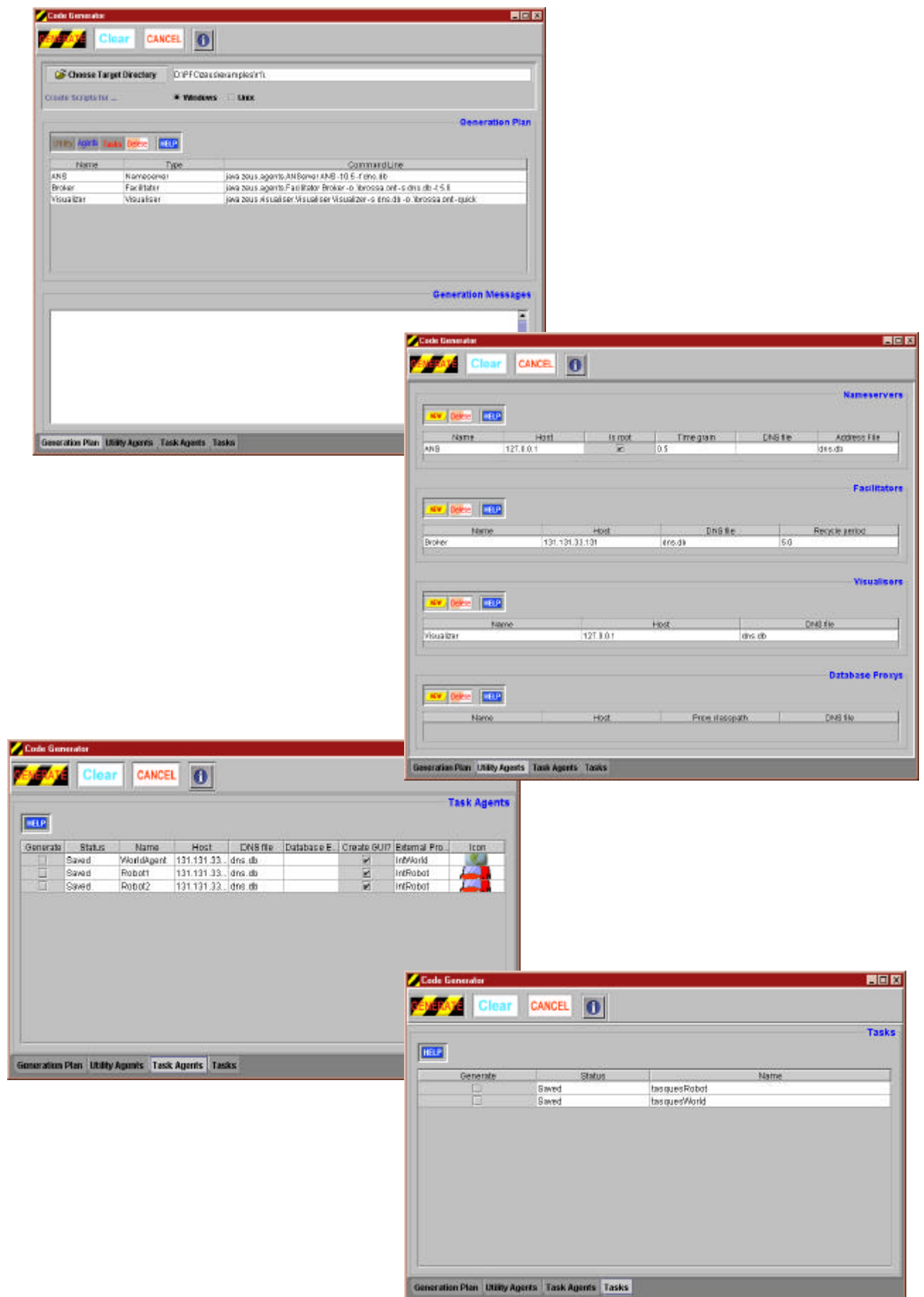


Figura 11.29: Panells inicialitzats per tal de generar el codi dels agents del sistema.

11.4.1.3. Funcionament dels agents

Tal i com s'ha dissenyat l'entorn, l'usuari és qui dona les ordres per tal que s'iniciï la recollida de la brossa per part dels robots.

Una vegada l'usuari dona l'ordre de generar una brossa, es genera aleatòriament la posició i s'avalua quin dels robots hi ha d'anar. El món té coneixement de la posició que ocupa cadascun d'ells. Llavors mira quin robot és més proper, i a aquest se li dona l'ordre de recollida de la nova brossa.

Un pas previ a iniciar el funcionament de la recollida és la inicialització de les posicions inicials dels robots i la presentació que fan al món. Aquesta presentació es fa notificant al món qui són (nom) i on són (columna i fila).

Aquest pas s'ha automatitzat per evitar conflictes de funcionament. Primer s'inicia l'agent món i després els robots, llavors el món va rebent les presentacions dels robots.

Aquesta manera de treballar és molt útil perquè podrien iniciar-se robots en diferents instants de temps i el món treballar amb els robots que té disponibles en cada moment, de manera que quan arriba un de nou, s'afegeix la seva localització i en la següent petició de recollida ja pot tenir-se en compte.

El funcionament de la comunicació de les posicions de les brosses i la presentació que fan els robots es veu en detall en l'apartat següent d'aquesta secció.

Per mantenir el control dels robots i assignar feina al robot que és més proper s'utilitza una classe creada expressament. Aquesta classe és *TaulaRobots* definida al fitxer *brossa.java*. Les interfícies gràfiques dels agents de l'aplicació fan referència a aquest fitxer per implementar les funcions.

És una estructura senzilla utilitzant tres vectors: dos vectors d'enters per la columna i la fila, i un vector d'strings (cadena de caràcters) pel nom.

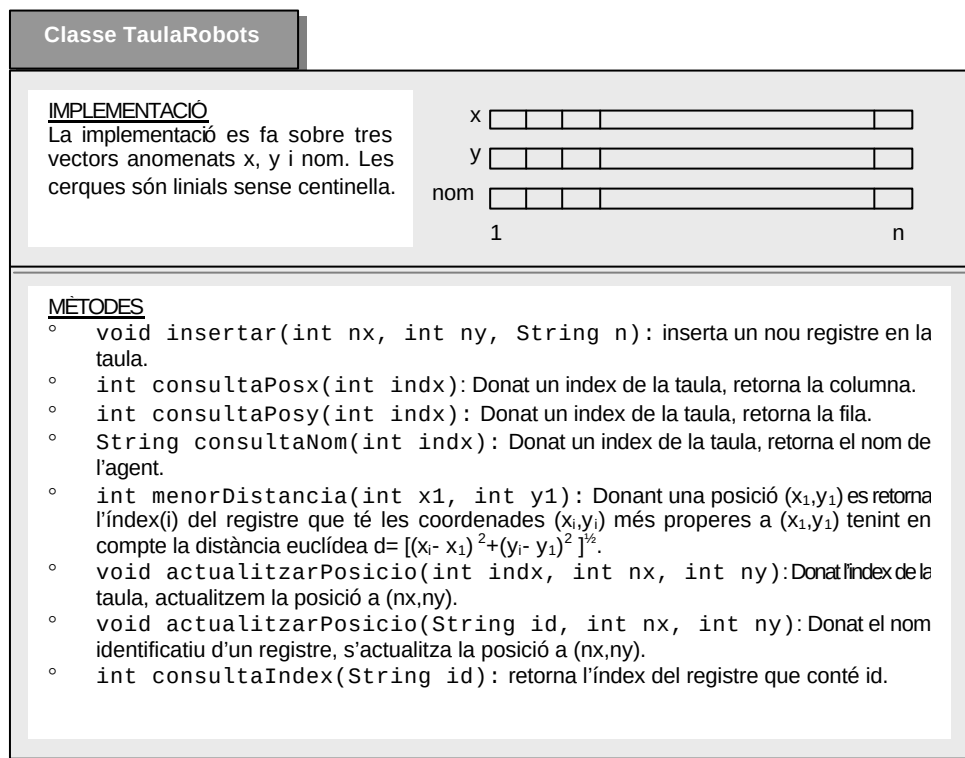


Figura 11.30: Descripció de la classe TaulaRobots.

11.4.1.4. Comunicació

La comunicació és el punt més important de l'aplicació, i alhora ha estat el punt més conflictiu del desenvolupament per la falta d'informació que proporciona Zeus.

La base de tot el conflicte és la forma de transmetre informació entre agents, com es poden transmetre dades i la manera de processar aquesta informació?. És un punt que Zeus ha de solucionar en futures versions.

Així per tal de comunicar-se s'ha hagut de muntar una petita estructura.

Hi ha un punt important de Zeus del qual encara no se n'ha parlat que és la definició de l'ontologia de l'aplicació. L'ontologia forma part dels elements per comunicar-se.

L'ontologia és l'entorn que forma la Base de Coneixement dels agents. L'ontologia està constituïda per variables i fets que estan formats per estructures de dades que poden estar compostes d'un o més elements simples (enters, caràcters, ...) o d'elements compostos (v.Figura 11.31).

La diferència entre variables (*entity*) i fets (*abstract*) és la utilitat i ús que tenen. Tant les variables com els fets són estructures de dades. Els fets són instàncies locals de cada agent i poden utilitzar-se en tasques de tipus *rulebase* (regla de comportament); en canvi les variables, poden usar-se per proporcionar recursos inicials als agents i es fan servir per tasques de tipus *primitive* (regles de tipus precondition-acció).

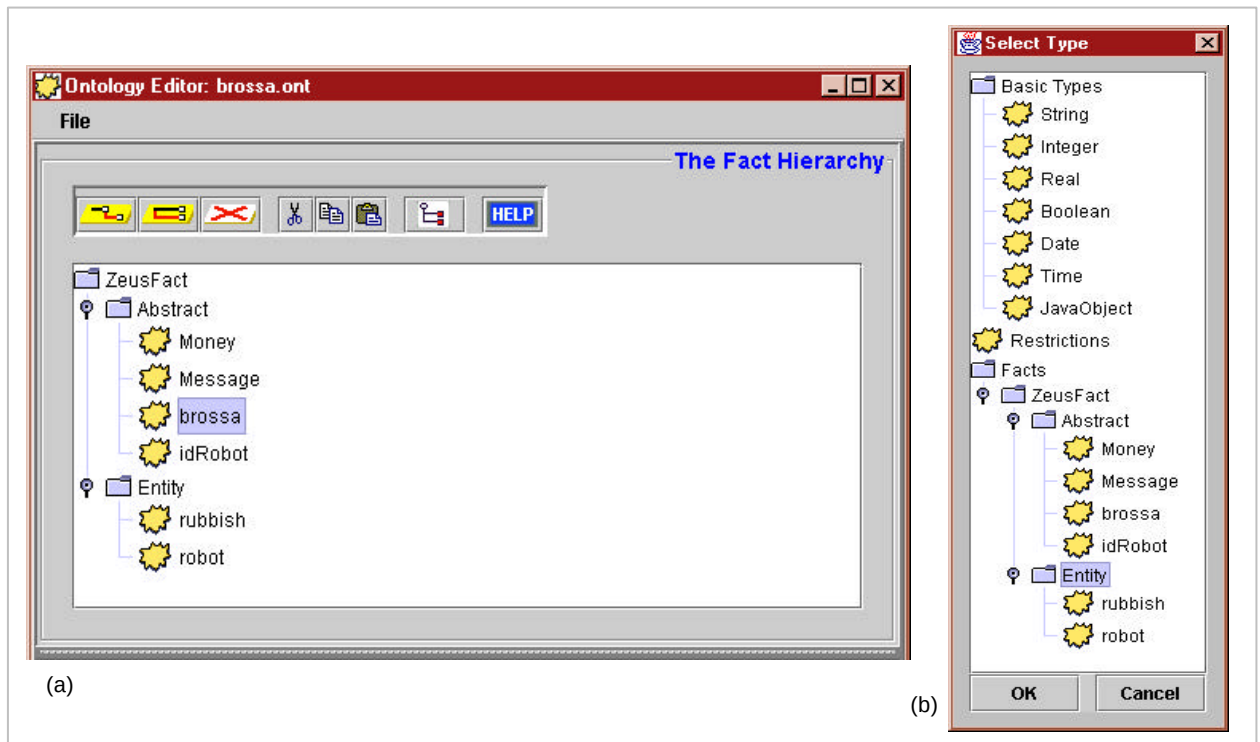
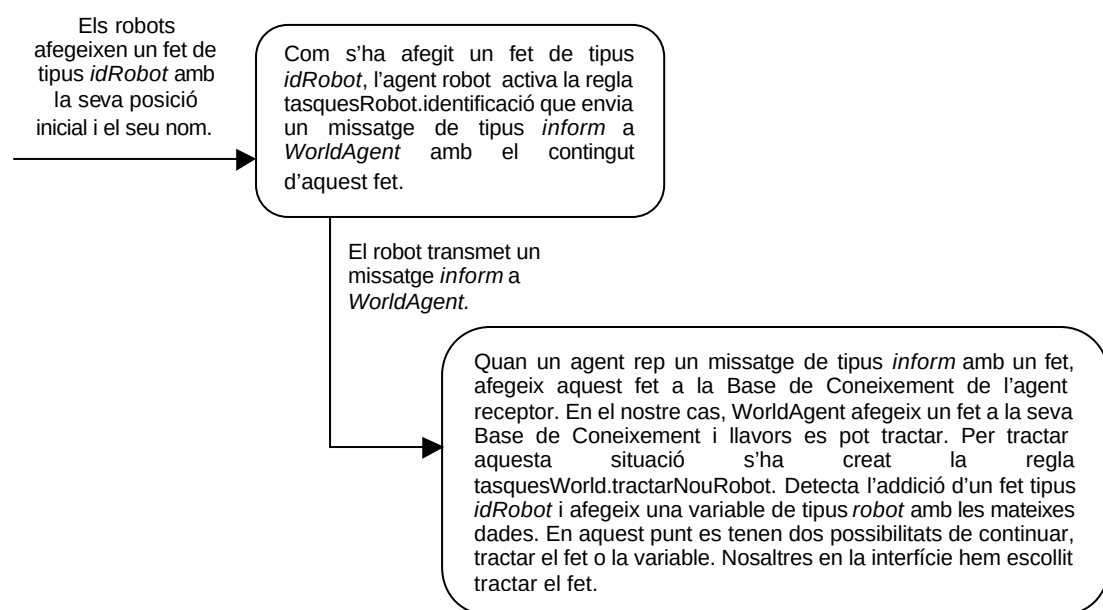
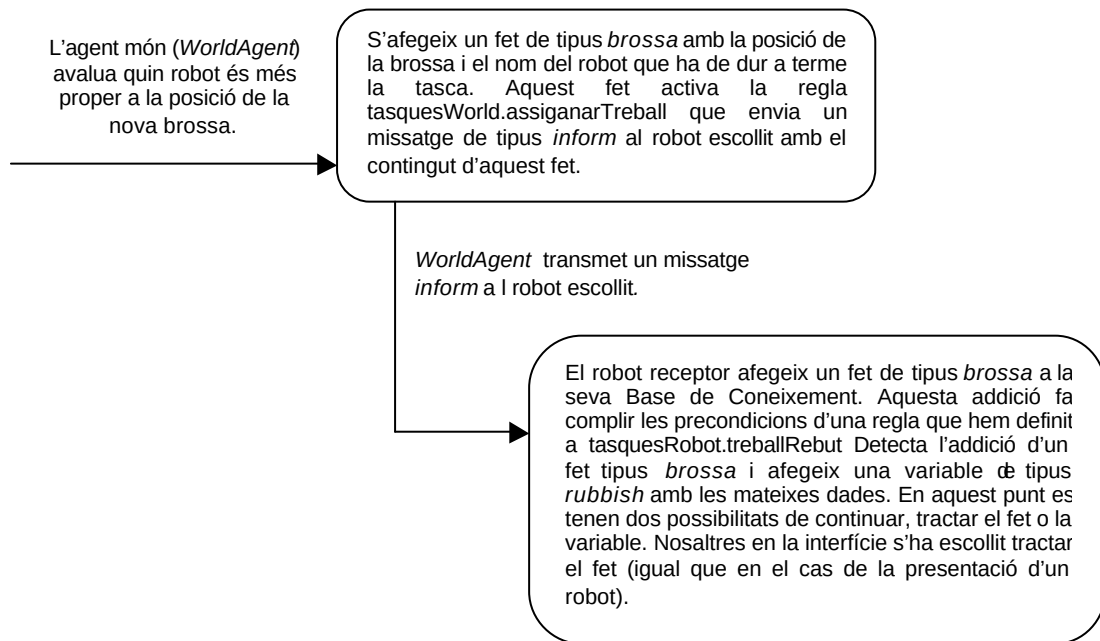


Figura 11.31: a) Estructura de variables i fets que tenim en la nostra aplicació.
b) Tipus de dades que podem escollir per definir camps d'aquestes variables i fets.

Les comunicacions que s'estableixen entre els agents de l'aplicació són de dos tipus diferents: presentació i assignació de feina.

PRESENTACIÓ





Veiem el codi de les tasques que realitzen els robots i el món. El codi que es mostra el genera l'entorn Zeus. Manté similituds amb llenguatges de Sistemes Experts, amb la notació precondició => acció.

```

(:Rulebase tasquesRobot
  (treballRebut
    (brossa (x_position ?x) (y_position ?y))
    =>
    (assert (rubbish (x_position ?x) (y_position ?y)))
  )
  (identificacio
    ?id <- (idRobot (nom ?n) (x_position ?x) (y_position ?y))
    =>
    (send_message (receiver WorldAgent) (content ?id) (type inform))
    (retract ?id)
  )
)

(:Rulebase tasquesWorld
  (assigantarTreball
    ?b <- (brossa (nom ?nomagent) (x_position ?x) (y_position ?y))
    =>
    (send_message (receiver ?nomagent) (content ?b) (type inform))
    (retract ?b)
  )
  (tractarNouRobot
    (idRobot (nom ?nom) (x_position ?x) (y_position ?y))
    =>
    (assert (robot (nom ?n) (x_position ?x) (y_position ?y)))
  )
)
  
```

Una de les dificultats per establir la comunicació va ser descobrir aquesta funció dels missatges del tipus *inform* que afegeix funcions diferents a les habituals d'altres sistemes.

La falta d'informació és evident en aquest tema, perquè no es proporciona informació de les funcions i quins tipus de missatges/accions es poden realitzar. Cal, però, tenir en compte que aquesta és la primera versió del producte i la documentació adjunta encara no és completa.

En la comunicació entre agents entra en joc l'agent ANS del sistema. Quan s'activa alguna de les regles que tenen com a acció a realitzar el *send_message*, com es veu a les regles, només es dóna el nom de l'agent receptor.

Les interfícies gràfiques que proporciona Zeus permeten d'observar quan i com es van succeint els missatges entre els agents (v.Creació dels agents). Es va observar el funcionament del sistema i es va observar com l'acció *send_message* demana a l'ANS informació del receptor, i quan realment l'emissor verifica que el receptor forma part del sistema i l'ANS té coneixement de la seva localització, li pot enviar el missatge (v. Figura 11.32).

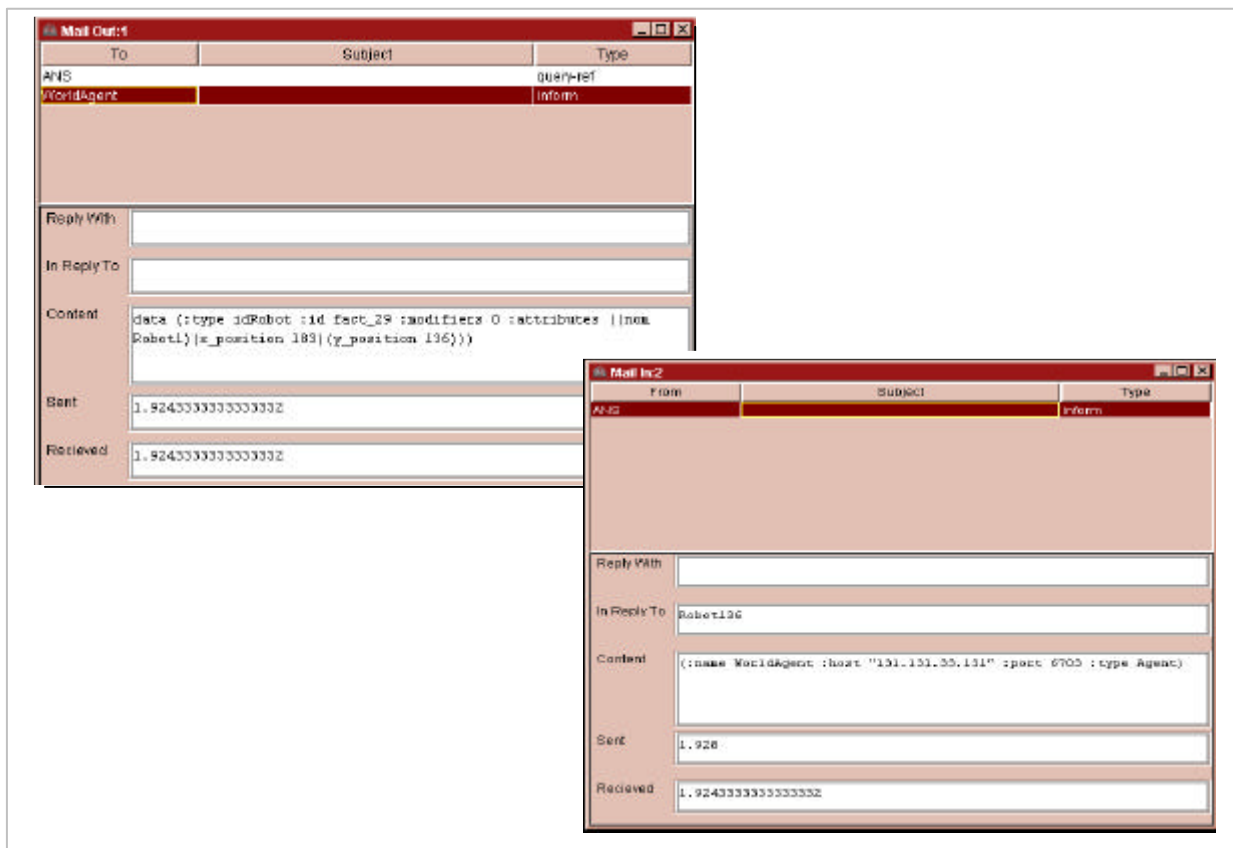


Figura 11.32: Missatges entrants i sortints de *Robot1* abans de rebre cap feina. Aquesta fase és la presentació del robot a *WorldAgent* i notifica la seva posició inicial. Vègis com *Robot1* demana (*query-ref*) l'adreça de *WorldAgent* a *ANS*. *Robot1* envia la seva presentació després de rebre la informació requerida d'*ANS*.

11.4.1.5. Resultats obtinguts

Els resultats obtinguts són bons. En la Figura 11.33 es mostra la comunitat d'agents que es tenen en la nostra aplicació.

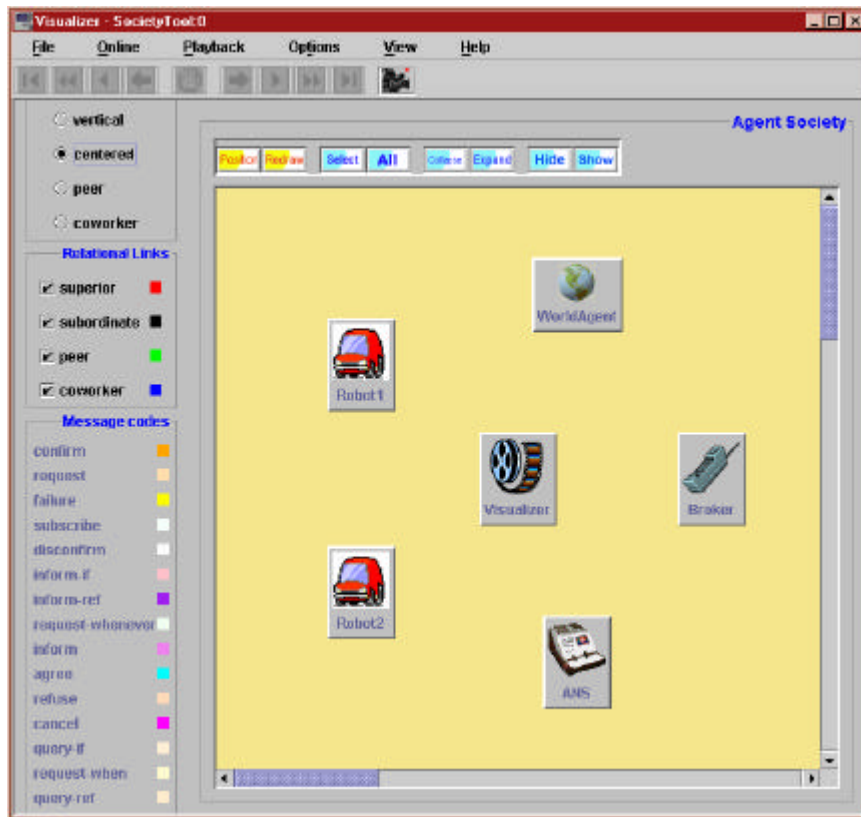


Figura 11.33: Comunitat d'agents de l'aplicació.

A banda dels agents pròpiament dits, en el nostre cas es té una interfície gràfica per cada agent de l'aplicació. El fet de tenir-les fa molt fàcil anar veient què està passant al sistema i quina és la situació de cada agent. Les àrees de text que tenen, mostren les tasques que van realitzant.

Abans de veure resultats, veiem l'aspecte d'aquestes interfícies i quins són els elements que les componen. Hi ha dos tipus diferents: un tipus pels agents robots, i un altre per l'agent món.

INTERFÍCIE DE L'AGENT ROBOT: ROBOT1 I ROBOT2.



Les interfícies dels robots tenen dues parts principals: una àrea de text on es van mostrant tots els missatges que calgui, i un panell amb informació de la posició que ocupen, la distància recorreguda en l'última feina i la posició de l'última brossa que han anat a recollir.

INTERFÍCIE DE L'AGENT MÓN: WORLDAGENT.



La interfície del món té com a finalitat que l'usuari vagi determinant quan s'han de crear les brosses, i vegi que es va produint.

Es mostra la posició de la brossa que s'ha generat i en l'àrea de text podem veure què va fent en cada moment.

En la figura es pot veure com inicialment els robots es *presenten* a l'agent món i aquest confirma la recepció d'aquestes notificacions.

El número de brosses que es poden generar és il·limitat. És l'usuari qui, tancant les finestres dels agents dóna per finalitzada la simulació.

Per veure el funcionament del sistema s'ha reproduït el text que es va mostrant en les diferents àrees de text per algunes brosses:

WorldAgent	Robot1	Robot2
WorldAgent espera instruccions...	Robot1 espera instruccions...	Robot2 espera instruccions...
S'ha identificat un nou Robot: Nom: Robot2 Posicio inicial: (31,142)	Creem nou fet: [idRobot] Robot1 es presenta a WorldAgent	Creem nou fet: [idRobot] Robot2 es presenta a WorldAgent
S'ha identificat un nou Robot: Nom: Robot1 Posicio inicial: (135,173)		
Creem nou fet: [brossa] Brossa a (151,197) Robot triat: Robot1 a (135,173) dist=28.844410203711913	Posició brossa (x,y): (151,197) ->Robot1: de (135,173) a (151,197)	Posició brossa (x,y): (54,131) ->Robot2: de (31,142) a (54,131)
Creem nou fet: [brossa] Brossa a (141,87) Robot triat: Robot1 a (135,173) dist=86.20904824900923	Posició brossa (x,y): (141,87) ->Robot1: de (151,197) a (141,87)	Posició brossa (x,y): (8,69) ->Robot2: de (54,131) a (8,69)
Creem nou fet: [brossa] Brossa a (199,107) Robot triat: Robot1 a (135,173) dist=91.93475947648963	Posició brossa (x,y): (199,107) ->Robot1: de (141,87) a (199,107)	Posició brossa (x,y): (48,112) ->Robot2: de (8,69) a (48,112)
Creem nou fet: [brossa] Brossa a (54,131) Robot triat: Robot2 a (31,142) dist=25.495097567963924	Posició brossa (x,y): (154,103) ->Robot1: de (199,107) a (154,103)	Posició brossa (x,y): (45,131) ->Robot2: de (48,112) a (45,131)
Creem nou fet: [brossa] Brossa a (199,107) Robot triat: Robot1 a (135,173) dist=91.93475947648963		
Creem nou fet: [brossa] Brossa a (8,69) Robot triat: Robot2 a (31,142) dist=76.53757247260981		
Creem nou fet: [brossa] Brossa a (48,112) Robot triat: Robot2 a (31,142) dist=34.48187929913333		
Creem nou fet: [brossa] Brossa a (45,131) Robot triat: Robot2 a (31,142) dist=17.804493814764857		
Creem nou fet: [brossa] Brossa a (154,103) Robot triat: Robot1 a (135,173) dist=72.53275122315436		

11.4.2. Codi de l'aplicació

El codi que es reproduïx fa referència a la interfície gràfica que s'ha creat. El codi dels agents i tasques es genera automàticament per Zeus i no cal reproduir-lo.

11.4.2.1. Interfície WorldAgent

La forma de la interfície i els mètodes a implementar s'especifiquen en la documentació que proporciona Zeus. Cal, però, adequar l'entorn a la nostra aplicació.

```
import zeus.actors.*;
import zeus.agents.*;

public class IntWorld implements ZeusExternal {

    public void exec(AgentContext agent) {
        Brossa monWin = new Brossa(agent,true);
    }

    public void showMsg(String message) {
    }

    public static void main(String[] args) {
        IntWorld win = new IntWorld();
        win.exec(null);
    }
}
```

11.4.2.2. Interfície Robot

És el mateix que en el cas de la interfície de *WorldAgent*. La diferència està en el pas de missatges cap a la classe principal de les interfícies (brossa), a la qual s'especifica si l'agent que implementarà és o no del tipus món, per així adequar les interfícies a les funcions que han de realitzar.

```
import zeus.actors.*;
import zeus.agents.*;

public class IntRobot implements ZeusExternal {

    public void exec(AgentContext agent) {
        Brossa brossaWin = new Brossa(agent,false);
    }

    public void showMsg(String message) {
    }

    public static void main(String[] args) {
        IntRobot win = new IntRobot();
        win.exec(null);
    }
}
```

11.4.2.3. Brossa

Cos principal de les interfícies. Segons sigui el cas d'un robot o no, es realitzen unes o altres tasques.

```
/**
// File: d:\pfc\zeus\examples\r1\brossa.java
// Programa principal de les interfícies dels robots i del mon.
//

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.border.*;
import java.lang.Thread;

import zeus.util.*;
import zeus.concepts.*;
import zeus.gui.fields.*;
import zeus.actors.*;
import zeus.actors.event.*;
import zeus.generator.util.*;

public class Brossa extends JFrame implements WindowListener {

    protected JPanel panellSimulacio;
    protected JTabbedPane tabbedPane;
    public JTextArea infoArea;
    public AgentContext agent;
    public JScrollPane scrollPane;
    public boolean esMon;
    public TaulaRobots taulaRobots;

    public Brossa(AgentContext ac,boolean esWorld) {
        agent = ac;
        esMon=esWorld;
        agent.ResourceDb().addFactMonitor(new SymFactMonitor(), FactEvent.ADD_MASK |
FactEvent.DELETE_MASK);
        if (esMon) { taulaRobots = new TaulaRobots(); }
        generarInterficie(esMon);
        pack();
        setResizable(true);
        show();
    }

    public void display(String message)
    {
        infoArea.append(message + "\n");
        infoArea.setCaretPosition(infoArea.getDocument().getLength());
    }

    protected class SymFactMonitor extends FactAdapter {
        public void factAddedEvent(FactEvent e) {
            Fact f = e.getFact();
            //display("S'ha activat factAddedEvent ...");

            if (f.getType().equals("brossa")) {
                int x = f.getInt("x_position");
            }
        }
    }
}
```

```

        int y = f.getInt("y_position");
        if (!esMon) display("Posició brossa (x,y): (" + x + ", " + y + ")");
        simPanel tmpPanel;
        tmpPanel = (simPanel)panellSimulacio;
        tmpPanel.escriureValorsPosicioBrossa(x,y);
        if(!esMon) { tmpPanel.recollirNovaBrossa(x,y); }
    }

    if(f.getType().equals("idRobot")) {
        if(esMon) {
            int x = f.getInt("x_position");
            int y = f.getInt("y_position");
            String nom = f.getValue("nom");
            taulaRobots.insertar(x,y,nom);
            display("S'ha identificat un nou Robot:");
            display(" Nom: " + nom + " Posicio inicial: (" + x + ", " + y + ")");
        }
        else {
            display(agent.whoami() + " es presenta a WorldAgent\n");
        }
    }
}

public void factDeletedEvent(FactEvent e) {
}

}

public void generarInterficie(boolean esMon) {
    String titol,nomIcona,nomImatge;
    if(esMon)
    {
        titol= "Generar brossa: " + agent.whoami();
        nomIcona = "gifs" + System.getProperty("file.separator") + "view.gif";
        nomImatge = "gifs" + System.getProperty("file.separator") + "Head3.gif";
    }
    else
    {
        titol= agent.whoami();
        nomIcona = "gifs" + System.getProperty("file.separator") + "van.gif";
        nomImatge = "gifs" + System.getProperty("file.separator") + "Head4.gif";
    }

    setTitle(titol);
    addWindowListener(this);

    getContentPane().setBackground(java.awt.Color.lightGray);
    ImageIcon icon = new ImageIcon(nomIcona);
    setIconImage(icon.getImage());

    GridBagLayout gridBagLayout = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    getContentPane().setLayout(gridBagLayout);
    getContentPane().setBackground(Color.lightGray);

    JLabel header = new JLabel(new ImageIcon(nomImatge)) {
        public void paint(Graphics g) {
            Dimension size = getSize();
            g.setColor(java.awt.Color.white);
            g.fillRect(0, 0, size.width, size.height);
            getIcon().paintIcon(this, g, 0, 0);
        }
    };
    header.setPreferredSize(new Dimension(418,133));
    header.setBorder(new BevelBorder(BevelBorder.RAISED));

    gbc.gridwidth = GridBagConstraints.REMAINDER;

```

```

gbc.anchor = GridBagConstraints.NORTH;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = gbc.weighty = 0;
gbc.insets = new Insets(3,3,3,3);
((GridBagLayout)getContentPane().getLayout()).setConstraints(header, gbc);
getContentPane().add(header);

infoArea = new JTextArea(7,58);
infoArea.setForeground(Color.black);
infoArea.setBackground(Color.white);
infoArea.setEditable(false);
infoArea.setText(agent.whoami() + " espera instrucciones...\n");
infoArea.setCaretPosition(infoArea.getDocument().getLength());
scrollPane = new JScrollPane(infoArea);

gbc.anchor = GridBagConstraints.WEST;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = gbc.weighty = 0;
gbc.insets = new Insets(3,3,3,3);
((GridBagLayout)getContentPane().getLayout()).setConstraints(scrollPane, gbc);
getContentPane().add(scrollPane);

tabbedPane = new JTabbedPane();
String msg;
if (esMon) {
    panellSimulacio = new simPanel(true,this,taulaRobots);
    msg="Generar brossa";
}
else {
    panellSimulacio = new simPanel(false,this,taulaRobots);
    msg="Recollir";
}

tabbedPane.addTab(msg, panellSimulacio);
tabbedPane.setTabPlacement(JTabbedPane.TOP);

gbc.anchor = GridBagConstraints.SOUTH;
gbc.fill = GridBagConstraints.BOTH;
gbc.weightx = gbc.weighty = 1;
gbc.insets = new Insets(8,8,8,8);
((GridBagLayout)getContentPane().getLayout()).setConstraints(tabbedPane, gbc);
getContentPane().add(tabbedPane);

}

public void windowClosed(WindowEvent event) {}
public void windowDeiconified(WindowEvent event) {}
public void windowIconified(WindowEvent event) {}
public void windowActivated(WindowEvent event) {}
public void windowDeactivated(WindowEvent event) {}
public void windowOpened(WindowEvent event) {}
public void windowClosing(WindowEvent event) {
    this.setVisible(false);
    this.dispose();
    System.exit(0);
}

}

//*****
class TaulaRobots {

    int i=1;
    static final int INICI=1;
    static final int N=11;

```

```

int x[] = new int[N];
int y[] = new int[N];
String nom[] = new String[N];

public void insertar(int nx, int ny, String n) {
    x[i]=nx;
    y[i]=ny;
    nom[i]=n;
    //System.out.println("Insertat: "+nom[i]+"("+x[i]+","+y[i]+")");
    i++;
}

public int consultaPosx(int indx) {
    return (int)x[indx];
}
public int consultaPosy(int indx) {
    return (int)y[indx];
}
public String consultaNom(int indx) {
    return (String)nom[indx];
}

public double calcularDist(int x0, int y0, int x1, int y1) {
    int a = x0-x1;
    int b = y0-y1;
    double d = (double)Math.sqrt(a*a+b*b);
    //System.out.println("Càlcul distància: "+d);
    return d;
}

public int menorDistancia(int x1, int y1) {
    System.out.println("Anem a buscar el robot que estigui més aprop de ("
    +x1+","+y1+")");
    int indx=INICI;
    double dmin=999999;
    for(int j=INICI; j<i; j++) {
        double dtemp = calcularDist(x1,y1,x[j],y[j]);
        System.out.println("Registre "+j+" ["+nom[j]+"] ("
        +x[j]+","+y[j]+"): d="+dtemp+" dmin="+dmin+" indx="+indx);
        if(dtemp<dmin) { dmin=dtemp; indx=j; }
    }
    return (int)indx;
}

public void actualitzarPosicio(int indx, int nx, int ny) {
    x[indx]=nx;
    y[indx]=ny;
}

public void actualitzarPosicio(String id, int nx, int ny) {
    int j=INICI;
    boolean trobat=false;
    while(!trobat && j<i) {
        if(nom[j].equals(id)) {
            actualitzarPosicio(j,nx,ny);
            trobat = true;
        }
        j++;
    }
}

public int consultaIndex(String id) {
    int indx=-1;
    int j=INICI;
    boolean trobat=false;
    while(!trobat && j<i) {

```

```

        if(nom[j].equals(id)) { indx=j; }
        j++;
    }
    return (int)indx;
}

}

//*****
class simPanel extends JPanel implements ActionListener {

    protected WholeNumberField labelPosx;
    protected WholeNumberField labelPosy;
    protected WholeNumberField labelPosxRobot;
    protected WholeNumberField labelPosyRobot;
    protected WholeNumberField labelDist;
    protected JButton tradeBtn;
    protected Brossa UI;
    public int rubx,ruby;
    public int robx,roby;
    protected boolean esMon;
    protected GridBagConstraints gbc;
    protected GridBagLayout gridBagLayout;
    public TaulaRobots t;

    public simPanel(boolean b, Brossa taulerPrincipal, TaulaRobots taula)
    {
        esMon = b;
        UI = taulerPrincipal;
        t = taula;

        gridBagLayout = new GridBagLayout();
        gbc = new GridBagConstraints();
        setLayout(gridBagLayout);
        setPreferredSize(new Dimension(360,220));
        setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));

        JPanel daltPanel = new JPanel();
        if(!esMon) {
            gbc.anchor=GridBagConstraints.WEST;
            gbc.fill=GridBagConstraints.BOTH;
            gbc.insets = new Insets(2,2,2,2);
            gbc.gridwidth = GridBagConstraints.REMAINDER;
            gbc.weightx = gbc.weighty = 1;
            gridBagLayout.setConstraints(daltPanel,gbc);
            add(daltPanel);
        }

        JPanel baixPanel = new JPanel();
        gbc.anchor=GridBagConstraints.EAST;
        gbc.fill=GridBagConstraints.BOTH;
        gbc.insets = new Insets(2,2,2,2);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.weightx = gbc.weighty = 1;
        gridBagLayout.setConstraints(baixPanel,gbc);
        add(baixPanel);

        if(!esMon) composarPanell1(daltPanel);

        composarPanell2(baixPanel);

        if (!esMon) { beginRobot("idRobot"); }
    }
}

```

```

public void posarTitolPanel(JPanel p, String msg) {
    TitledBorder border = BorderFactory.createTitledBorder(msg);
    border.setTitlePosition(TitledBorder.TOP);
    border.setTitleJustification(TitledBorder.RIGHT);
    border.setTitleFont(new Font("Helvetica", Font.BOLD, 10));
    border.setTitleColor(Color.blue);
    p.setBorder(border);
}

public void insertarRegistre(JPanel p, String msg, WholeNumberField etiNum) {

    JLabel titol = new JLabel(msg);
    titol.setFont(new Font("Helvetica",Font.PLAIN, 11));
    gbc.insets = new Insets(4,5,4,8);
    gbc.gridwidth = 1;
    gbc.fill = GridBagConstraints.NONE;
    gbc.anchor = GridBagConstraints.WEST;
    gridBagLayout.setConstraints(titol,gbc);
    p.add(titol);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.fill = GridBagConstraints.NONE;
    gbc.insets = new Insets(4,8,4,15);
    gridBagLayout.setConstraints(etiNum, gbc);
    p.add(etiNum);
}

public void composarPanell1(JPanel p) {
    posarTitolPanel(p,"ROBOT");

    labelPosxRobot = new WholeNumberField(0, 999);
    labelPosxRobot.setBackground(Color.white);
    labelPosxRobot.setPreferredSize(new Dimension(50,20));
    labelPosxRobot.setValue(0);

    labelPosyRobot = new WholeNumberField(0, 999);
    labelPosyRobot.setBackground(Color.white);
    labelPosyRobot.setPreferredSize(new Dimension(50,20));
    labelPosyRobot.setValue(0);

    labelDist = new WholeNumberField(0, 9999);
    labelDist.setBackground(Color.lightGray);
    labelDist.setPreferredSize(new Dimension(50,20));
    labelDist.setValue(0);

    insertarRegistre(p,"Columna[x]: ",labelPosxRobot);
    insertarRegistre(p,"Fila[y]: ",labelPosyRobot);
    insertarRegistre(p,"Distància: ",labelDist);
}

public void composarPanell2(JPanel p) {

    posarTitolPanel(p,"BROSSA");

    labelPosx = new WholeNumberField(0, 999);
    labelPosx.setBackground(Color.white);
    labelPosx.setPreferredSize(new Dimension(50,20));
    labelPosx.setValue(0);

    labelPosy = new WholeNumberField(0, 999);

```

```

        labelPosy.setBackground(Color.white);
        labelPosy.setPreferredSize(new Dimension(50,20));
        labelPosy.setValue(0);

        JPanel p1 = new JPanel();
        JPanel p2 = new JPanel();
        p1.setPreferredSize(new Dimension(360,80));
        p2.setPreferredSize(new Dimension(360,80));
        p.add(p1);
        p.add(p2);
        insertarRegistre(p1,"Columna[x]:",labelPosx);
        insertarRegistre(p1,"Fila[y]: ",labelPosy);

        if (esMon) {
            gbc.gridwidth = GridBagConstraints.REMAINDER;
            gbc.fill = GridBagConstraints.NONE;
            gbc.insets = new Insets(8,8,8,8);
            gbc.weightx = 1;
            tradeBtn = new JButton("Simulacio");
            tradeBtn.addActionListener(this);
            gridBagLayout.setConstraints(tradeBtn,gbc);
            p2.add(tradeBtn);
        }
    }

    public void recollirNovaBrossa(int x, int y) {
        UI.display("->" + UI.agent.whoami() + ": de (" + robx + ", " + roby + ") a (" + x + ", " + y + ") \n");

        int dist = (int) calcularDist(x, y, robx, roby);
        escriureValorsPosicioRobot(x, y, dist);
        setRobot(x, y);
        int indx = t.consultaIndex(UI.agent.whoami());
        if (indx > 0) { t.actualitzarPosicio(indx, x, y); }
    }

    public double calcularDist(int x0, int y0, int x1, int y1) {
        int a = x0 - x1;
        int b = y0 - y1;
        double d = (double) Math.sqrt(a * a + b * b);
        return d;
    }

    public void escriureValorsPosicioBrossa(int x, int y) {
        labelPosx.setValue(x);
        labelPosy.setValue(y);
    }

    public void escriureValorsPosicioRobot(int x, int y, int d) {
        labelPosxRobot.setValue(x);
        labelPosyRobot.setValue(y);
        labelDist.setValue(d);
    }

    public void setRobot(int x, int y) {
        robx = x;
        roby = y;
    }

    public int getRobx() {
        return robx;
    }

    public int getRoby() {
        return roby;
    }

```

```

public void iniPosicioRobot() {
    Random rand = new Random ((long)System.currentTimeMillis());
    int MAX = 200;
    robx = Math.abs(rand.nextInt()%MAX);
    roby = Math.abs(rand.nextInt()%MAX);
}

public void actionPerformed(ActionEvent evt)
{
    if (UI.agent == null ) return;
    if (evt.getSource() == tradeBtn) {
        if (esMon) beginDemo("brossa");
    }
}

public void beginDemo(String tipus)
{
    UI.display("\nCreem nou fet: ["+tipus+"]");
    if (tipus.equals("brossa")) {
        boolean acabat=false;
        while(!acabat) {
            Fact brossa = UI.agent.OntologyDb().getFact(Fact.FACT, tipus);
            Random rand = new Random ((long)System.currentTimeMillis());
            int MAX = 200;
            rubx = Math.abs(rand.nextInt()%MAX);
            ruby = Math.abs(rand.nextInt()%MAX);
            brossa.setValue("x_position",rubx);
            brossa.setValue("y_position",ruby);

            labelPosx.setValue(rubx);
            labelPosy.setValue(ruby);

            int idx = t.menorDistancia(rubx,ruby);
            int x = t.consultaPosx(idx);
            int y = t.consultaPosy(idx);
            String nomAgentTriat = t.consultaNom(idx);
            double d = calcularDist(x,y,rubx,ruby);
            System.out.println("Registre triat: ["+nomAgentTriat+"]
("+x+", "+y+") d="+d);
            System.out.println("");
            brossa.setValue("nom",nomAgentTriat);

            UI.display("Brossa a ("+rubx+", "+ruby+")");
            UI.display("Robot triat: "+nomAgentTriat+" a ("+x+", "+y+")
dist="+d);

            escriureValorsPosicioBrossa(rubx,ruby);

            UI.agent.ResourceDb().add(brossa);
            acabat=true;
        }
    }
}

public void beginRobot(String tipus)
{
    Thread t = new Thread();
    try {
        Thread.sleep(10000);
    } catch (InterruptedException ee) { }

    UI.display("\nCreem nou fet: ["+tipus+"]");
    boolean acabat=false;
    while(!acabat) {
        iniPosicioRobot();
        int x=getRobx();
        int y=getRoby();

```

```

        escriureValorsPosicioRobot(x,y,0);

Fact posRobot = UI.agent.OntologyDb().getFact(Fact.FACT, tipus);
posRobot.setValue("x_position",x);
posRobot.setValue("y_position",y);
posRobot.setValue("nom",UI.agent.whoami());
UI.agent.ResourceDb().add(posRobot);
acabat=true;
    }
}
}

```

11.5. Aplicacions pràctiques i conclusions

L'entorn de desenvolupament de Zeus facilita el ràpid desenvolupament d'aplicacions basades en agents que poden requerir de col·laboració entre ells. Es donen les llibreries i un entorn visual, i es donen facilitats per tal d'afegir prestacions que ens calguin fer servir.

L'entorn està pensat per poder-se adequar a molts dominis de treball i s'ha dissenyat en dues parts: disseny i execució/compilació. D'aquesta manera s'espera que l'etapa de disseny es faci estàndard per qualsevol aplicació. Més enllà, la compilació i execució de les aplicacions es faran sobre les diferents màquines, però la utilització del llenguatge de programació Java fa que les diferències s'anulin.

L'entorn de desenvolupament Zeus s'ha provat en aplicacions de comerç electrònic (simulacions) tenint diferents estratègies de coordinació.

La cooperació i negociació de Zeus s'han implementat en una aplicació que simula una cadena de muntatge (en aquest cas d'ordinadors).

Podem pensar en dominis en els quals es pot fer servir aquest entorn:

- Sistemes de coneixement distribuït. L'ontologia permet de distribuir *facts* dins els missatges, i es poden processar.
- Interfícies home-computadora. L'aplicació d'exemple es pot considerar dins d'aquesta categoria.
- Assistents personals. Aquests sistemes treballen amb i per a l'usuari. Podem pensar en agents dispersats per la xarxa i capaços de recollir informació. És una aplicació complicada pensant en les poques facilitats que dona Zeus pensant en *www*, *html*, *email*, etc. Hauríem d'implementar algunes coses.
- Comerç electrònic. Amb reserves donat que no es proporcionen eines de seguretat en transaccions comercials. No s'implementa cap tipus de codificació, signatura, etc. Així aquestes aplicacions quedarien en la fase de simulació.
- Serveis compra/venda. Passa el mateix que el cas de comerç electrònic.
- Serveis financers.
- Automatització d'oficines. Es poden generar tasques planificades, així com regles de comportament automatitzades.
- Organització d'empreses. Podem pensar en l'automatització d'una factoria (cadena de muntatge).
- Gestió de robots.

Com s'ha vist en aquest capítol, Zeus trenca la tendència dels entorns estudiats fins ara, i proporciona tot un entorn de desenvolupament, amb eines per crear agents, executar-los, monitoritzar-los i fins i tot per treure estadístiques de l'execució que s'hagi realitzat. Es nota que els creadors d'aquesta eina han estat anys investigant i provant en entorns d'agents, i s'intenta donar una resposta a aquestes inquietuds.

L'usuari sempre agraeix un entorn visual per interactuar amb el sistema. Fins ara es donaven els ingredients i era el programador qui havia de crear l'aplicació; ara l'aplicació es crea sola i el programador pot dedicar-se a polir detalls o a millorar algunes funcions.

El sistema Zeus que s'ha provat és la primera versió que es difon d'aquest entorn; encara no està acabada i falten algunes coses per finalitzar. El sistema té unes icones que l'usuari pot pitjar per demanar ajuda del panell o zona on sigui, però la quasi majoria d'ajudes no estan encara implementades. Zeus pensa incorporar més funcions a l'agent *visualiser* i que es pugui intervenir sobre la definició dels agents (tasca que es realitza a *ZEUS Agent Generator*), però aquestes funcions encara no estan implementades. Tampoc no es proporciona la documentació d'un tercer exemple molt útil, proporcionat amb el sistema.

KAFKA: Yet Another Multi-Agent Library for Java (Kafka)

Fujitsu Corporation

<http://www.fujitsu.co.jp/hypertext/free/kafka/>



David Isern Alarcón, aisern@tinet.fut.es
Enginyeria Tècnica Informàtica de Sistemes

Professor: Antoni Moreno, URV, amoreno@etse.urv.es

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV), <http://www.etse.urv.es>

Curs 1998-99

12. ESTUDI DE KAFKA	212
12.1. DESCRIPCIÓ GENERAL.....	214
12.2. METODOLOGIA DE DESENVOLUPAMENT	215
12.2.1. <i>Eines oferides per Kafka</i>	215
12.2.1.1. Acció get	215
12.2.1.2. Acció set	217
12.2.1.3. Acció call	218
12.2.1.4. Acció eval	218
12.2.1.5. Acció moveTo	220
12.2.1.6. Accions del Directory	220
12.2.2. <i>Procés de disseny</i>	221
12.3. DESENVOLUPAMENT D'UNA APLICACIÓ.....	221
12.3.1. <i>Descripció de l'aplicació</i>	221
12.3.1.1. Directori d'agents	221
12.3.1.2. Creació dels agents.....	222
12.3.1.3. Funcionament dels agents	223
12.3.1.4. Comunicació	225
12.3.1.5. Resultats obtinguts.....	227
1.1.1.1.1. Simulació de l'aplicació	228
12.3.2. <i>Codi complet</i>	232
12.3.2.1. Robots	232
1.1.1.1.2. Classe robot	232
1.1.1.1.3. Acció remota dels robots: AccioRemota	233
12.3.2.2. Agent world.....	235
12.3.2.3. Tauler de robots de l'agent món	237
12.3.2.4. Interfície de world	238
12.4. APLICACIONS.....	240

12.1. Descripció general

Kafka és una llibreria de classes dissenyada per a construir SMA basant-se en aplicacions distribuïdes. Kafka és flexible, extensible i fàcil de fer servir per programadors que estan avesats a realitzar aplicacions distribuïdes en Java.

Es poden donar diferents característiques per aquest software:

- **Reacció davant esdeveniments.** Els agents poden modificar el seu comportament (amb codis de programes) en temps d'execució. El comportament dels agents és representat per una classe d'accions abstractes (*Action*). És molt útil per els manteniments remots o per instal·lacions de serveis.
- **Avaluació remota.** Els agents poden rebre i avaluar codis de programes (classes) amb o sense objectes fent servir els mètodes proporcionats per RMI.
- **Directorí de noms.** Els agents poden no estar localitzats físicament a la mateixa màquina, i cal algun element que tingui informació de les localitzacions.
- **Polítiques de seguretat.** Flexibles, modelables i aplicades a diferents nivells de seguretat.
- **100% java i compatible amb RMI.** Kafka està escrit completament en Java. Els agents integren Java RMI; els agents poden comunicar-se directament utilitzant la transmissió d'objectes RMI.
- **Simplicitat i facilitat d'aprenentatge.** Tot hi haver-hi unes llibreries amb algunes classes i bastants mètodes, no és difícil aprendre com s'utilitzen. A més, es proporcionen forces exemples d'aplicació específica d'eines que aclareixen moltes dubtes.

Característiques de KAFKA	
Llenguatge de programació	Java (JDK 1.1)
Cada agent treballa per sí sol?	No, tot i que s'implemeta una classe Agent aquesta és una extensió de la classe <i>Object</i> de Java.
Comunicació entre agents	Crides de mètodes, i transferència de dades/objectes utilitzant la classe RMI.
ANS	Mantingut pel <i>kafka-directory</i> .
És escalable?	Si
ACL	ACL definit per l'usuari.
Proporciona conceptes de coordinació?	No
Característiques de seguretat	Kafka proporciona eines de mobilitat d'agents, i per tant implementa eines de seguretat en diferents nivells.

12.2. Metodologia de desenvolupament

La metodologia de disseny és igual a la metodologia de desenvolupament d'Aplicacions Orientades a Objectes (AOO).

12.2.1. Eines oferides per Kafka

Les eines de les llibreries Kafka que podem fer servir estan pensades per a dues finalitats: mètodes cridats desde qualsevol agent, i distribució/moviment de dades a través dels agents.

Abans de dissenyar qualsevol tipus d'aplicació cal tenir presents quines eines ofereix Kafka per a ser utilitzades. Bàsicament, es poden dividir en quatre tipus: *set/get*, *call*, *eval*, *moveTo* i *Directory*.

12.2.1.1. Acció get.

Els agents tenen assignats diferents atributs (similars a variables locals) que poden ser consultats sense cap problema per qualsevol agent (v.Taula 12.1).

Atribut	Descripció	Exemple
local.host	Dóna l'adreça local de la màquina	localhost/127.0.0.1
boot.host	Dóna l'adreça d'on és l'agent	localhost/127.0.0.1
global.name	Nom de l'agent complet. Com es local no hi ha més paràmetres.	world.da2bbb8297
user.tmp	Directory on és	/mnt/d/pfc/kafka/cop_ex/prova
boot.time	Hora d'inici de l'execució de l'agent	Tue Sep 07 19:28:45 GMT+02:00 1999
name	Nom de l'agent	world.da2bbb8297
user.name	Nom de l'usuari que executa l'agent	root

Taula 12.1: Atributs dels agents disponibles per defecte. Per consultar el seu valor cal fer `get("nom_atribut")`

Aquests atributs poden ser consultar-se amb l'acció `get`. Aquestes crides estan subjectes a controls de seguretat.

Aquests atributs poden ampliar-se segons les necessitats de l'agent. Els atributs que podem afegir poden ser de qualsevol tipus de dades definits per Java, o fins i tot codi Java executable. En aquest últim l'atribut afegit es considerarà com acció.

Ara bé, quan nosaltres afegim algun altre atribut (comanda `add`), un dels paràmetres de l'assignació és un *control d'accés* (un codi identificatiu de les operacions permeses sobre aquell atribut) i l'accés serà controlat.

Abans de veure un exemple de la utilització de la comanda `get`, veiem com funciona el *control d'accés*. El paquet `kafka.AccessMode` ofereix diferents constants per cada tipus d'operació permesa segons el propietari de l'atribut vulgui (v.Taula 12.2).

// Access scope		
public final static byte	GLOBAL	= 0;
public final static byte	DOMESTIC	= 1;
public final static byte	LOCAL	= 2;
// Writability mode		
public final static short	GLOBAL_WRITABLE	= 0002;
public final static short	DOMESTIC_WRITABLE	= 0020;
public final static short	LOCAL_WRITABLE	= 0200;
// Read only mode		
public final static short	GLOBAL_CONSTANT	= 0444;
public final static short	DOMESTIC_CONSTANT	= 0440;
public final static short	LOCAL_CONSTANT	= 0400;
// Locally writable mode		
public final static short	GLOBAL_ACCESS	= 0644;
public final static short	DOMESTIC_ACCESS	= 0640;
public final static short	LOCAL_ACCESS	= 0600;
// Executability mode		
public final static short	GLOBAL_CALL	= 0711;
public final static short	DOMESTIC_CALL	= 0710;
public final static short	LOCAL_CALL	= 0700;

Taula 12.2: Modes d'accés a atributs/accions definits per Kafka

En la Figura 12.1 es pot veure un exemple d'utilització de la comanda `get`.

```
import kafka.*;
import kafka.agent.*;

public class TestAgent extends Agent {
    public TestAgent(String name) {
        add("married", new Boolean(true), GLOBAL_ACCESS);
        add("age", new Integer(31), LOCAL_ACCESS);
        add("access test", new LocalClassData("TestAction"), GLOBAL_CALL);
    }
}

public class TestAction extends Action {
    protected Object start(Object arg[]) {
        try {
            System.out.println("married = " + self().get("married"));
        } catch (Exception e) { System.out.println(e); }
        try {
            System.out.println("age = " + self().get("age"));
        } catch (Exception e) { System.out.println(e); }
        return null;
    }
}
```

Figura 12.1: Kafka proporciona eines per assignar atributs als agents i accions que poden ser avaluades remotament. Es tenen eines de seguretat per controlar l'accés a aquestes dades, i és l'usuari qui ho determina.

Llavors, segons qui cridi l'acció pot donar error. Si qui crida l'acció "access test" és cridada per l'agent propietari, donarà:

```
married = true
age = 34
```

Però si l'acció es crida des d'un altre agent (utilitzant una comanda que més endavant veurem `-eval()`), la crida no s'executa amb èxit perquè l'atribut *age* només pot ser accedit localment.

12.2.1.2. Acció set

S'ha vist que els agents tenen assignats una sèrie d'atributs que poden ser consultats per altres agents. També s'ha vist que els agents no només tenen assignats els atributs per defecte, sinó que l'usuari pot definir-ne de nous. Doncs bé, la comanda `set` de Kafka permet assignar valors a aquests atributs en temps d'execució.

Per exemple, si nosaltres afegim un nou atribut "test" de tipus *LocalClassData* (accions creades/propietat de l'agent), farem:

```
robot()
{
    super("Robot", System.getProperty("user.dir"));
    add("test", new LocalClassData("AccioRemota"), GLOBAL_CALL);
}
```

i s'afegeix el codi `GLOBAL_CALL` que permet a qualsevol agent executar l'acció *test* però no permet de ser modificada.

Una vegada afegit aquesta acció a l'agent, si es requereix, es pot assignar en temps d'execució algun valor determinat (per exemple, si el valor de l'atribut ha de canviar segons el comportament de l'aplicació). Per assignar un cert valor, es fa servir la comanda/acció `set`:

```
ClassData method = new LocalClassData("AccioRemota");
set("test",method);
```

L'assignació d'un valor a una acció amb la comanda `set` està supeditada als permisos que hagi donat el propietari de l'agent a aquest atribut.

12.2.1.3. Acció call

L'acció `call` realitza la crida d'una acció amb o sense arguments. L'acció ha de ser del tipus `ClassData` o `ClassArchive` i ha de tenir els permisos per a poder ser executada. Després de l'execució, l'acció pot retornar algun tipus de valor.

El funcionament és similar a l'execució d'un procediment amb uns paràmetres d'entrada i un codi de retorn.

La classe `ClassData` permet de llegir codi de programes en Java, i la classe `ClassArchive` permet llegir fitxers en general, sense haver de ser codi.

Aquesta comanda ha estat implementada en l'aplicació d'exemple, per la qual cosa, si es vol més informació del seu funcionament es pot veure la descripció que se'n fa a l'apartat 12.3.1.4.

12.2.1.4. Acció eval

L'acció `eval` té la mateixa funció que la funció `call` però en varia la forma d'executar-se.

Aquesta acció enlloc de cridar un mètode d'un agent, agafa i envia el codi del mètode que vol executar, l'executa a l'agent destí i es retorna el resultat.

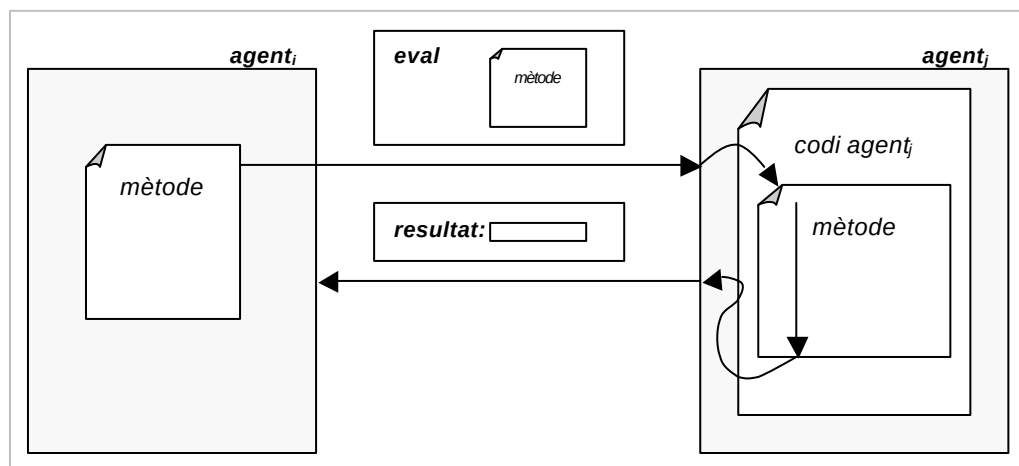


Figura 12.2: Execució d'una comanda `eval`. El codi del mètode a executar es transporta ("si tú no pots anar a la muntanya, la muntanya ve a tú").

Aquesta comanda té un perill important donat que un agent pot rebre qualsevol tipus de codi, i si no es prenen mesures, pot passar alguna desgràcia. Kafka ha previst aquestes eventualitats, i dins de les mesures de seguretat, s'implementen una sèrie de mètodes (que poden ser reescrits per l'usuari) per *chequejar* les accions que s'executen remotament.

La classe *kafka.agent.AccessControl* implementa aquests mètodes que no només inclouen l'acció *eval*, sinó que inclou totes les accions que tenen algun perill (v.Taula 12.3).

Acció	Descripció
<code>public void checkAdd(Remote requester, String name, Object val, int mode) throws AccessException</code>	Mirem si un nou atribut o acció pot ser afegida especificant el nom, valor i mode d'accés.
<code>public void checkRemove(Remote requester, String name) throws AccessException</code>	Mirem si l'atribut o acció amb el nom especificat pot ser esborrada de l'agent.
<code>public void checkSet(Remote requester, String name, Object val) throws AccessException</code>	Mirem si l'atribut o acció amb el nom especificat poden ser modificats amb el nou valor.
<code>public void checkGet(Remote requester, String name) throws AccessException</code>	Mirem si el valor actual de l'atribut o acció amb el nom especificat poden ser obtinguts.
<code>public void checkCall(Remote requester, String name) throws AccessException</code>	Mirem si l'acció amb el nom especificat pot ser executada.
<code>public void checkEval(Remote requester, byte code[], String name) throws AccessException</code>	Mirem si l'acció que ha estat enviada pot executar-se.

Taula 12.3: Mètodes implementats per *kafka.agent.AccessControl* per a chequejar si diverses operacions poden dur-se a terme de manera segura.

12.2.1.5. Acció moveTo

L'acció `moveTo` mou l'agent que l'executa al context d'execució d'un altre agent i reinicia la seva execució al destí.

L'agent abans de moure's fa un stop de la seva execució, i es transmet utilitzant la serialització implementada pel paquet RMI de Java.

12.2.1.6. Accions del Directory

Algunes accions precedents s'executen remotament i cal informació de l'agent destí. La informació dels agents està inclosa en dues classes *Agent* i *Remote*. La primera implementa mètodes per trobar la informació necessària dins de cada agent (com per exemple `self()`), i la segona és necessària per executar crides com `get`, `call`, o `eval`.

La inicialització d'una variable de tipus *Remote* requereix d'una estructura auxiliar que tingui coneixement de la localització de tots els agents. Aquesta estructura és la del paquet *kafka.agent.Directory*.

Dins de l'aplicació d'exemple s'utilitzen diferents mètodes, per exemple:

- per saber els noms de tots els agents que estan actius en el moment de fer la crida:
`String llistaNoms[] = directory.list();`
- per saber la localització d'un agent determinat. Es busca inicialment en el directori actual, i si el nom és el global, es mira tota la màquina.
`Remote ro = directory.lookup(nom);`
- Si sabem el nom d'un agent, es pot fer una cerca de si està o no actiu:
`Remote ro = directory.search(nom);`

12.2.2. Procés de disseny

El procés de disseny d'una aplicació Kafka pot fer-se igual que una Aplicació Orientada a Objectes (AOO).

Un possible procés seria:

- ➔ **Identificar classes (agents) i objectes (accions i atributs).** Cal pensar quins elements són representatius de l'aplicació quins serveis han d'implementar.
- ➔ **Identificar-ne la semàntica;** que volen dir cadascun dels elements.
- ➔ **Identificar les relacions entre ells;** crides, paràmetres, i valors de retorn.
- ➔ **Especificar les interfícies i la implementació d'aquestes classes i objectes;** cal pensar com es realitzaran les relacions.
- ➔ **Desenvolupar un model de la conducta desitjada del sistema (anàlisi):** identificar un programa principal que modeli/monitoritzi el funcionament.
- ➔ **Refinar reiteradament la implementació (disseny):** aquest procés no és linial, cal anar retocant detalls i fer voltes dins d'aquest procés.

12.3. Desenvolupament d'una aplicació

12.3.1. Descripció de l'aplicació

L'aplicació és la del *món de la brossa*.

En el nostre sistema es tenen dos tipus d'agents: el món i els robots. El món monitoritza l'execució del sistema i va generant periòdicament una brossa en una posició aleatòria. El món quan genera una nova brossa, mira quina agents hi ha actius al sistema i la seva posició, i després de mirar quin robot és el més proper, li envia la posició de la brossa per a recollir-la.

12.3.1.1. Directori d'agents

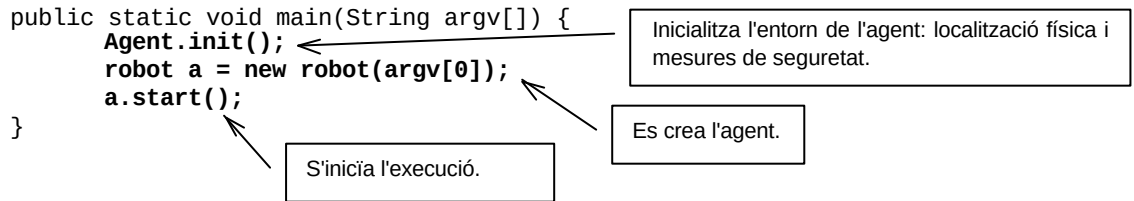
Els agents es registren en el directori d'agents del sistema. Recordem que abans d'iniciar qualsevol aplicació cal engegar el *directory* de Kafka:

```
$java kafka.agent.DirectoryImpl &
```

Una vegada engegats cadascun dels agents, aquests poden requerir serveis d'altres agents del sistema. Per tal de facilitar la comunicació entre els agents, el *directory* de Kafka proporciona una sèrie de mètodes (v. apartat 12.2.1.6).

12.3.1.2. Creació dels agents

El sistema Kafka proporciona una classe *Agent* per desenvolupar nous agents. Si robot és una extensió de la classe agent, per crear i executar-lo, fem:



L'inicialització de la classe de tipus *agent* (robot), pot implicar la creació d'altres operacions necessàries pel funcionament del sistema. Kafka permet d'executar accions/mètodes d'altres agents. Per fer-ho possible, els agents defineixen accions que poden ser accedides remòtament. Aquestes accions poden rebre i retornar paràmetres. En el nostre cas, cada robot té una acció que pot ser accedida remòtament. Aquesta acció té dues funcions: 1) proporcionar la posició actual del robot, i 2) actualitzar la posició del robot quan el món li ho requereix (v. Figura 12.6).

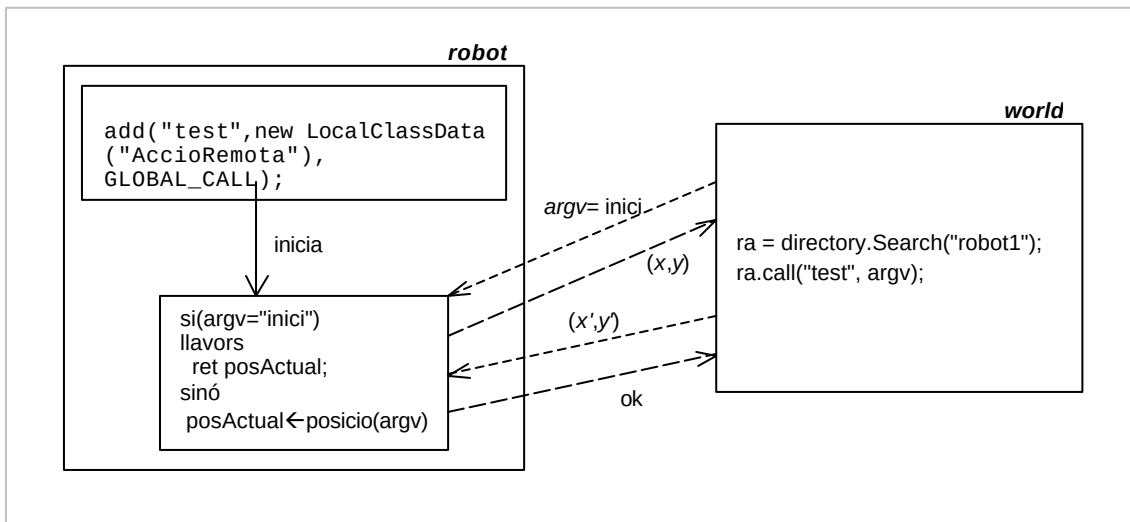


Figura 12.6: Passos de la inicialització dels elements dels robots. El mètode apodat test és accedit per *world*. L'agent *world* per accedir al mètode de robot ha de saber el seu nom i ha de buscar la seva localització física (variable ra).

12.3.1.3. Funcionament dels agents

El funcionament dels agents és senzill:

WORLD

L'agent *world* cada 5 segons inicia una nova simulació. Primer de tot mira quins agents hi ha actius al sistema. Per cadascun d'ells (menys ell mateix), fa una crida del mètode remot *test* del robot amb l'argument `arg[0]="inici"`. D'aquesta manera captura la posició actual del robot i insereix les seves dades a `tr` que és una variable de la classe *TaulaRobots*.(v. Taula 12.1).

Una vegada sabem la posició de tots els robots generem la posició d'una nova brossa. Utilitzant un mètode de la classe *TaulaRobots* (`tr#menorDistancia`), podem saber les dades del robot que és més proper a la brossa. Una vegada identificat es fa una crida del mètode *test* del robot escollit passant-li la posició de la brossa. El robot actualitza la seva posició.

ROBOT

Els agents robots estan esperant instruccions. Una vegada s'activa el seu mètode *test*, segons sigui el valor dels arguments d'entrada, es procedeix a realitzar les accions pertinents.

MÈTODES DE LA CLASSE TaulaRobots

- `void insertar(int nx, int ny, String n)`: inserta un nou registre en la taula.
- `int consultaPosx(int indx)`: Donat un index de la taula, retorna la columna.
- `int consultaPosy(int indx)`: Donat un index de la taula, retorna la fila.
- `String consultaNom(int indx)`: Donat un index de la taula, retorna el nom de l'agent.
- `int menorDistancia(int x1, int y1)`: Donant una posició (x_1, y_1) es retorna l'índex(i) del registre que té les coordenades (x_i, y_i) més properes a (x_1, y_1) tenint en compte la distància euclídea $d = [(x_i - x_1)^2 + (y_i - y_1)^2]^{1/2}$.
- `void actualitzarPosicio(int indx, int nx, int ny)`: Donat l'index de la taula, actualitzem la posició a (nx, ny) .
- `void actualitzarPosicio(String id, int nx, int ny)`: Donat el nom identificatiu d'un registre, s'actualitza la posició a (nx, ny) .
- `int consultaIndex(String id)`: retorna l'índex del registre que conté id.

Taula 12.1: Mètodes proporcionats per la classe *TaulaRobots*. És una classe feta a mida per les necessitats del problema.

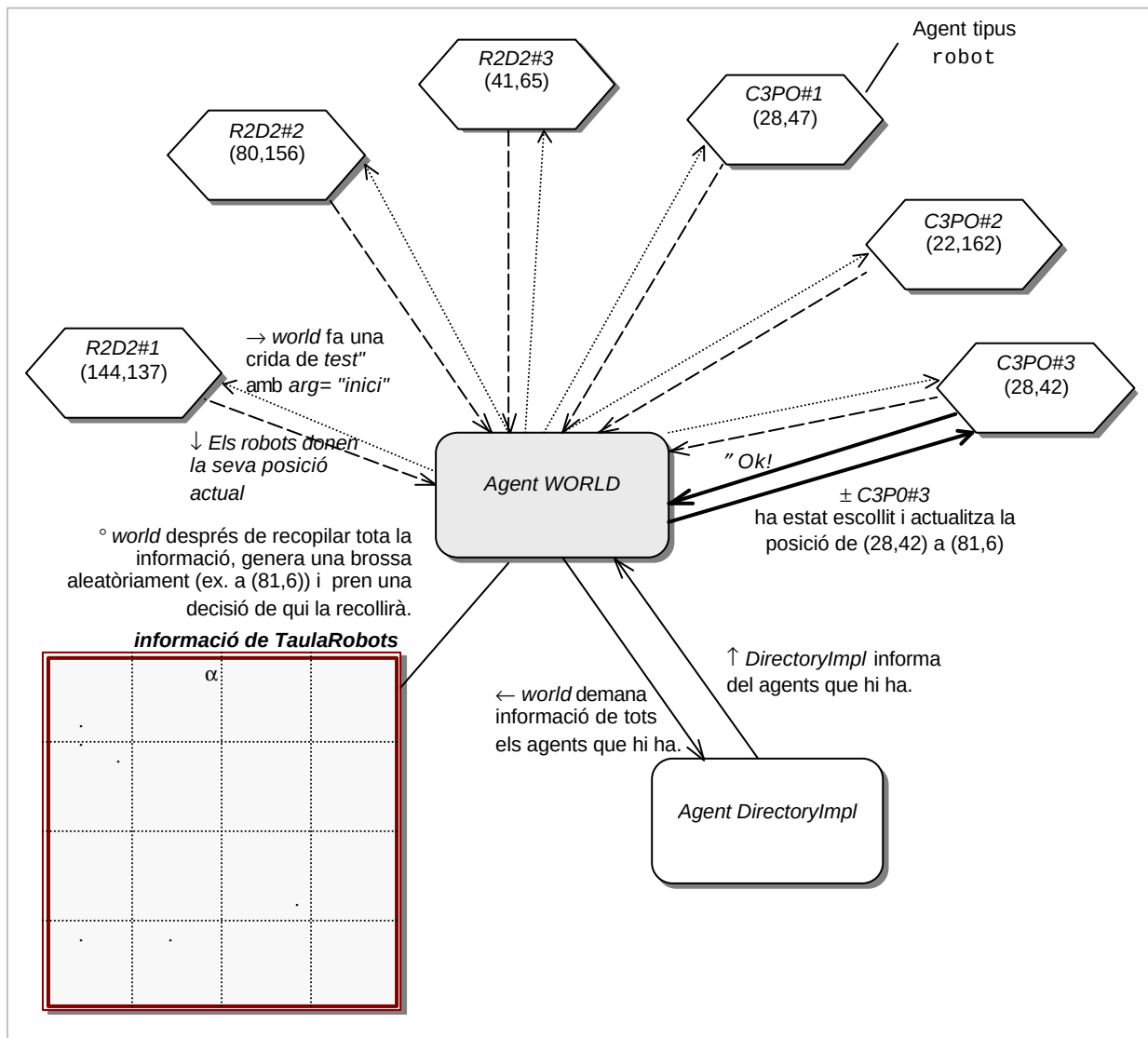


Figura 12.7: Esquema de les interaccions entre els diferents agents del sistema per tal d'aconseguir l'objectiu de recollir una brossa generada aleatòriament.

ACCÉS A LES DADES DELS AGENTS REMOTS

Els agents han de realitzar crides a accions que es troben en altres agents. Aquest fet implica que l'agent que vol iniciar la crida ha de saber la localització (host i port) del receptor. L'agent *DirectoryImpl* proporciona aquests serveis a la resta d'agents.

Per accedir a la llista d'agents de l'aplicació en el moment de fer la crida, es pot fer:

```
String llistaNoms[] = directory.list();
```

Aquesta crida només proporciona el nom/etiqueta que donem als agents. Per realitzar una crida d'un mètode remot, ens cal més informació de l'agent amb qui volem contactar i s'han d'aconseguir les seves dades de tipus *RemoteAgent*.

```
String nom=llistaNoms[i];
Remote ro = directory.lookup(nom);
RemoteAgent ra = (RemoteAgent)ro;

String cadRet=(String)ra.call("test", arg);
```

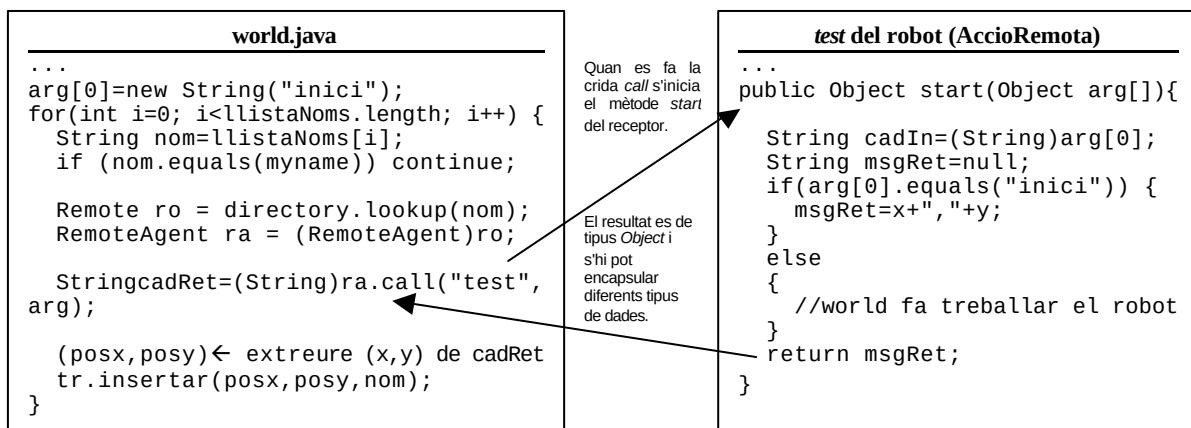
Com es veu, la variable *ra* conté més informació de l'agent i amb la qual podem fer el call.

12.3.1.4. Comunicació

La comunicació que s'estableix entre els agents està basada en crides a mètodes d'altres agents i, si és rellevant, avaluar el resultat d'aquesta crida.

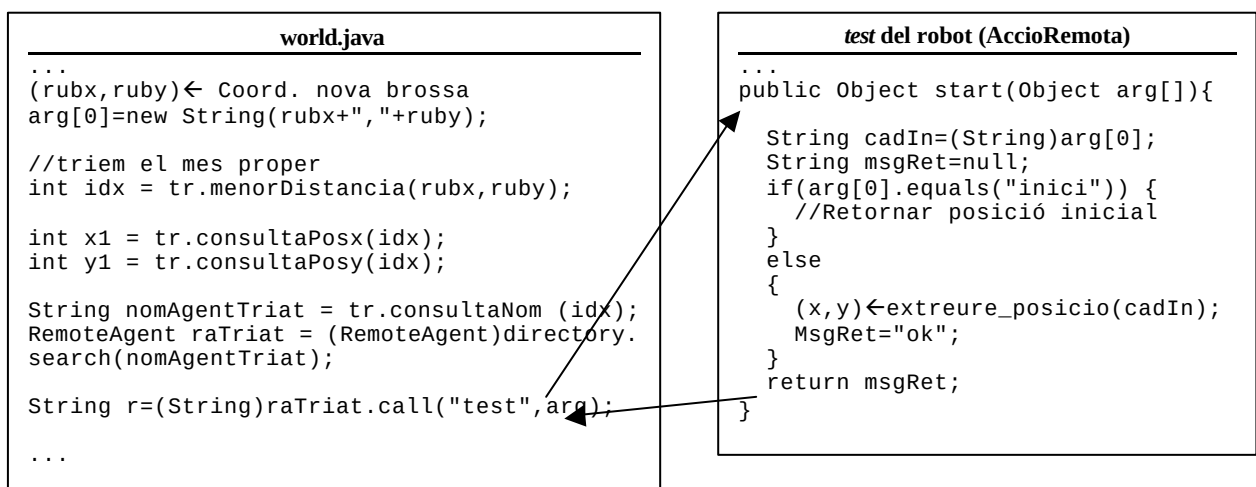
Les crides que es fa en la nostra aplicació són del tipus *call*, i es passen arguments i es té un codi de retorn de tipus *string*.

Quan world vol saber on es troba un agent robot, fa la crida *call* amb l'argument "inici", de manera que el robot que ho rep pot saber que fer. Veiem com es fa:



Com veiem el llenguatge de comunicació no es basa en ACLs definits i usats àmpliament. Els agents no es comuniquen a través de missatges, sinó que implementen eines per a dur a terme accions per les quals són requerits, i els agents són capaços d'executar remotament. Aquesta forma de treballar és molt semblant a la programació orientada a objectes.

L'altra funció que realitza *world* és assignar feina a un robot. Com s'ha vist, quan el robot rep una crida de l'acció *test* mira el valor dels paràmetres d'entrada. Si el paràmetre d'entrada és diferent d'"inici", es considera que és una crida de *world* passant la posició de la brossa que ha de recollir. Després d'extreure les coordenades (x_{rub}, y_{rub}) de la brossa, el robot actualitza la seva posició: $(x_{rob}, y_{rob}) \leftarrow (x_{rub}, y_{rub})$.



12.3.1.5. Resultats obtinguts

Els resultats són força bons. Per veure el funcionament de l'aplicació s'ha desenvolupat una interfície gràfica (v. Figura 12.8) per l'agent *world* que permet d'observar les evolucions.

La simulació consta de 20 creacions de noves brosses cada 5 segons (es podia haver fet de qualsevol altre manera), i cada cop l'agent *world* mostra informació relativa a allò que està fent i els resultats que són rellevants.

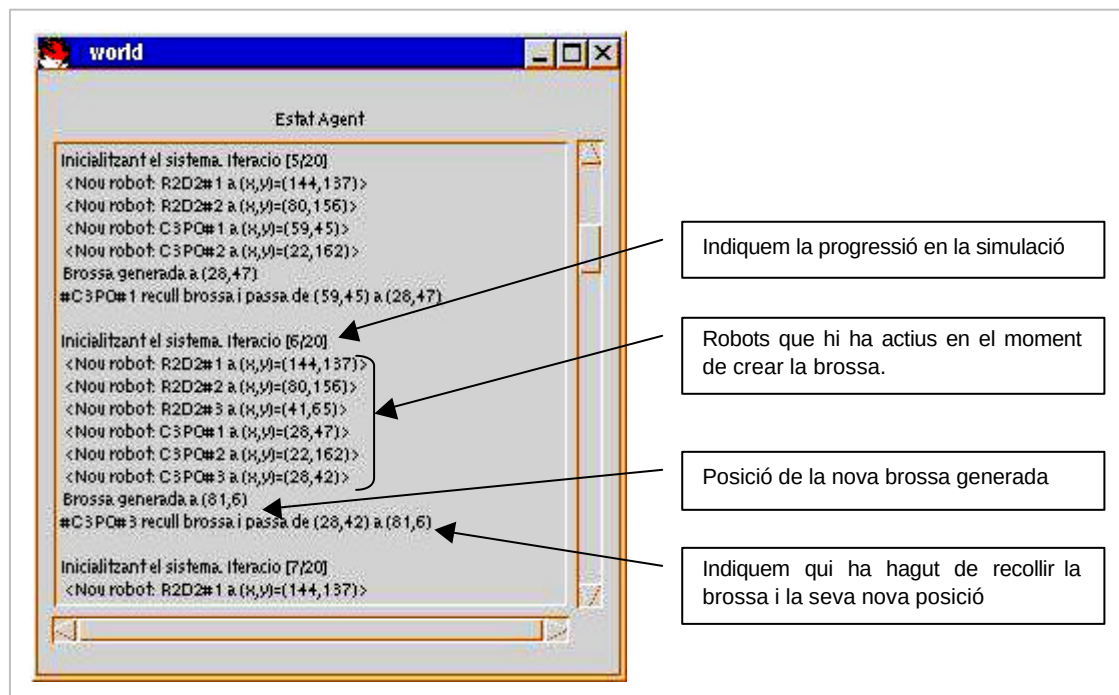


Figura 12.8: Interfície gràfica de l'agent *world*.

Per veure més bé el funcionament de l'aplicació s'han reproduït les evolucions d'una simulació. Es donen detalls del funcionament de *world* i dels diferents grups de robots que intervien.

1.1.1.1.1. Simulació de l'aplicació

Reproduïm els missatges que es van mostrant en l'execució de l'aplicació.

Resultats de world

En la simulació es van creant i destruint diversos robots. Cada vegada que es creen de nous es creen dos de cop, i igualment es destrueixen. En la simulació hi ha fins a 10 robots a la vegada (iteracions 9 i 10), i el sistema funciona correctament. La creació/destrucció dels diferents agents són voluntat de l'usuari.

```
Inicialitzant el sistema. Iteracio [1/20]
<Nou robot: R2D2#1 a (x,y)=(192,99)>
<Nou robot: C3P0#1 a (x,y)=(5,122)>
Brossa generada a (60,71)
#C3P0#1 recull brossa i passa de (5,122) a (60,71)

Inicialitzant el sistema. Iteracio [2/20]
<Nou robot: R2D2#1 a (x,y)=(192,99)>
<Nou robot: C3P0#1 a (x,y)=(60,71)>
Brossa generada a (166,110)
#R2D2#1 recull brossa i passa de (192,99) a (166,110)

Inicialitzant el sistema. Iteracio [3/20]
<Nou robot: R2D2#1 a (x,y)=(166,110)>
<Nou robot: C3P0#1 a (x,y)=(60,71)>
Brossa generada a (144,137)
#R2D2#1 recull brossa i passa de (166,110) a (144,137)

Inicialitzant el sistema. Iteracio [4/20]
<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: C3P0#1 a (x,y)=(60,71)>
Brossa generada a (59,45)
#C3P0#1 recull brossa i passa de (60,71) a (59,45)

Inicialitzant el sistema. Iteracio [5/20]
<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: C3P0#1 a (x,y)=(59,45)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
Brossa generada a (28,47)
#C3P0#1 recull brossa i passa de (59,45) a (28,47)

Inicialitzant el sistema. Iteracio [6/20]
<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(28,42)>
Brossa generada a (81,6)
#C3P0#3 recull brossa i passa de (28,42) a (81,6)

Inicialitzant el sistema. Iteracio [7/20]
<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: R2D2#4 a (x,y)=(52,73)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(81,6)>
<Nou robot: C3P0#4 a (x,y)=(180,176)>
Brossa generada a (125,40)
```

#C3P0#3 recull brossa i passa de (81,6) a (125,40)

Inicialitzant el sistema. Iteracio [8/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: R2D2#4 a (x,y)=(52,73)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(125,40)>
<Nou robot: C3P0#4 a (x,y)=(180,176)>

Brossa generada a (193,40)

#C3P0#3 recull brossa i passa de (125,40) a (193,40)

Inicialitzant el sistema. Iteracio [9/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: R2D2#4 a (x,y)=(52,73)>
<Nou robot: R2D2#5 a (x,y)=(39,35)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(193,40)>
<Nou robot: C3P0#4 a (x,y)=(180,176)>
<Nou robot: C3P0#5 a (x,y)=(8,87)>

Brossa generada a (56,82)

#R2D2#4 recull brossa i passa de (52,73) a (56,82)

Inicialitzant el sistema. Iteracio [10/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: R2D2#4 a (x,y)=(56,82)>
<Nou robot: R2D2#5 a (x,y)=(39,35)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(193,40)>
<Nou robot: C3P0#4 a (x,y)=(180,176)>
<Nou robot: C3P0#5 a (x,y)=(8,87)>

Brossa generada a (92,70)

#R2D2#4 recull brossa i passa de (56,82) a (92,70)

Inicialitzant el sistema. Iteracio [11/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: R2D2#4 a (x,y)=(92,70)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(193,40)>
<Nou robot: C3P0#4 a (x,y)=(180,176)>

Brossa generada a (162,75)

#C3P0#3 recull brossa i passa de (193,40) a (162,75)

Inicialitzant el sistema. Iteracio [12/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#2 a (x,y)=(80,156)>
<Nou robot: R2D2#3 a (x,y)=(41,65)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>
<Nou robot: C3P0#2 a (x,y)=(22,162)>
<Nou robot: C3P0#3 a (x,y)=(162,75)>

Brossa generada a (75,98)

#R2D2#3 recull brossa i passa de (41,65) a (75,98)

Inicialitzant el sistema. Iteracio [13/20]

<Nou robot: R2D2#1 a (x,y)=(144,137)>
<Nou robot: R2D2#3 a (x,y)=(75,98)>
<Nou robot: C3P0#1 a (x,y)=(28,47)>

<Nou robot: C3PO#3 a (x,y)=(162,75)>
 Brossa generada a (34,188)
#R2D2#3 recull brossa i passa de (75,98) a (34,188)

Inicialitzant el sistema. Iteracio [14/20]
 <Nou robot: R2D2#1 a (x,y)=(144,137)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (77,161)
#R2D2#1 recull brossa i passa de (144,137) a (77,161)

Inicialitzant el sistema. Iteracio [15/20]
 <Nou robot: R2D2#1 a (x,y)=(77,161)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (160,177)
#R2D2#1 recull brossa i passa de (77,161) a (160,177)

Inicialitzant el sistema. Iteracio [16/20]
 <Nou robot: R2D2#1 a (x,y)=(160,177)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (194,134)
#R2D2#1 recull brossa i passa de (160,177) a (194,134)

Inicialitzant el sistema. Iteracio [17/20]
 <Nou robot: R2D2#1 a (x,y)=(194,134)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (177,83)
#R2D2#1 recull brossa i passa de (194,134) a (177,83)

Inicialitzant el sistema. Iteracio [18/20]
 <Nou robot: R2D2#1 a (x,y)=(177,83)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (175,141)
#R2D2#1 recull brossa i passa de (177,83) a (175,141)

Inicialitzant el sistema. Iteracio [19/20]
 <Nou robot: R2D2#1 a (x,y)=(175,141)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (119,110)
#R2D2#1 recull brossa i passa de (175,141) a (119,110)

Inicialitzant el sistema. Iteracio [20/20]
 <Nou robot: R2D2#1 a (x,y)=(119,110)>
 <Nou robot: C3PO#1 a (x,y)=(28,47)>
 Brossa generada a (187,36)
#R2D2#1 recull brossa i passa de (119,110) a (187,36)

Evolucions de C3PO#1 i R2D2#1

C3PO#1 i R2D2#1 són vius durant tota la simulació. En aquesta simulació els dos robots treballen nombroses vegades.

Creem els agents...

Inicialitzem posicio. Posicio inicial (x,y)=(192,99)

Inicialitzem posicio. Posicio inicial (x,y)=(5,122)

Agents creats.

C3PO#1 brossa tractada: (x,y)=(5,122) a (x,y)=(60,71)
R2D2#1 brossa tractada: (x,y)=(192,99) a (x,y)=(166,110)
R2D2#1 brossa tractada: (x,y)=(166,110) a (x,y)=(144,137)
C3PO#1 brossa tractada: (x,y)=(60,71) a (x,y)=(59,45)
C3PO#1 brossa tractada: (x,y)=(59,45) a (x,y)=(28,47)
R2D2#1 brossa tractada: (x,y)=(144,137) a (x,y)=(77,161)
R2D2#1 brossa tractada: (x,y)=(77,161) a (x,y)=(160,177)
R2D2#1 brossa tractada: (x,y)=(160,177) a (x,y)=(194,134)
R2D2#1 brossa tractada: (x,y)=(194,134) a (x,y)=(177,83)

R2D2#1 brossa tractada: (x,y)=(177,83) a (x,y)=(175,141)
R2D2#1 brossa tractada: (x,y)=(175,141) a (x,y)=(119,110)
R2D2#1 brossa tractada: (x,y)=(119,110) a (x,y)=(187,36)

Evolucions de C3PO#2 i R2D2#2

C3PO#2 i R2D2#2 es creen després de C3PO#1 i R2D2#1. Són actius des de la simulació 5/20 fins la 12/20 quan són destruïts per l'usuari. En la simulació, aquests dos robots no treballen mai.

Creem els agents...
Inicialitzem posicio. Posicio inicial (x,y)=(80,156)

Inicialitzem posicio. Posicio inicial (x,y)=(22,162)

Agents creats.

Evolucions de C3PO#3 i R2D2#3

C3PO#3 i R2D2#3 es creen després de C3PO#2 i R2D2#2. Són actius des de la simulació 6/20 fins la 13/20 quan són destruïts per l'usuari. En la simulació, aquests dos robots treballen alguna vegada.

Creem els agents...
Inicialitzem posicio. Posicio inicial (x,y)=(41,65)

Inicialitzem posicio. Posicio inicial (x,y)=(28,42)

Agents creats.
C3PO#3 brossa tractada: (x,y)=(28,42) a (x,y)=(81,6)
C3PO#3 brossa tractada: (x,y)=(81,6) a (x,y)=(125,40)
C3PO#3 brossa tractada: (x,y)=(193,40) a (x,y)=(162,75)
R2D2#3 brossa tractada: (x,y)=(41,65) a (x,y)=(75,98)
R2D2#3 brossa tractada: (x,y)=(75,98) a (x,y)=(34,188)

Evolucions de C3PO#4 i R2D2#4

C3PO#4 i R2D2#4 es creen posteriorment a C3PO#3 i R2D2#3. Són actius des de la simulació 7/20 fins la 11/20 quan són destruïts per l'usuari. En la simulació només R2D2#4 treballa.

Creem els agents...
Inicialitzem posicio. Posicio inicial (x,y)=(52,73)

Inicialitzem posicio. Posicio inicial (x,y)=(180,176)

Agents creats.
R2D2#4 brossa tractada: (x,y)=(52,73) a (x,y)=(56,82)
R2D2#4 brossa tractada: (x,y)=(56,82) a (x,y)=(92,70)

Evolucions de C3PO#5 i R2D2#5

C3PO#5 i R2D2#5 són els últims robots en crear-se. Són actius des de la simulació 9/20 fins la 10/20 quan són destruïts per l'usuari. En tan pocs intents de recollir brossa, no els toca mai.

Creem els agents...
Inicialitzem posicio. Posicio inicial (x,y)=(39,35)

Inicialitzem posicio. Posicio inicial (x,y)=(8,87)

Agents creats.

12.3.2. Codi complet

12.3.2.1. Robots

Els robots es creen a partir d'un fitxer "mare" què és Agents.java. En un primer moment es van crear dos robots (R2D2 i C3PO) però per avaluar millor el funcionament de l'aplicació es poden crear dinàmicament robots. Per diferenciar cadascun dels robots, es pot passar com a paràmetre al fitxer un atribut que s'afegirà al nom. En el joc de proves s'han anat creant R2D2#1, R2D2#2, R2D2#3, etc. i igualment per C3PO.

```
//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/Agents.java

import kafka.*;
import kafka.agent.*;
import java.rmi.*;

public class Agents
{
    public static void main(String argv[])
        throws RemoteException, NotBoundException, AlreadyBoundException
    {
        Debug.debug = false;
        Agent.init();
        System.out.println("Creem els agents...");
        robot r1 = new robot("R2D2#" + argv[0]);
        robot r2 = new robot("C3PO#" + argv[0]);
        System.out.println("Agents creats.");
        r1.start();
        r2.start();
    }
}
```

Classe robot

Els diferents robots són de la classe robot:

```
//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/robot.java

import kafka.*;
import kafka.agent.*;
import java.rmi.*;
import java.util.Date;

public class robot extends Agent
{
    robot()
        throws RemoteException, NotBoundException, AlreadyBoundException
    {
        super("Robot", System.getProperty("user.dir"));
        add("test", new LocalClassData("AccioRemota"), GLOBAL_CALL);
    }

    robot(String name)
```

```

        throws RemoteException, NotBoundException, AlreadyBoundException
    {
        super(name, System.getProperty("user.dir"));
        add("test", new LocalClassData("AccioRemota"), GLOBAL_CALL);
    }

    public void run() {
        ClassData method = new LocalClassData("AccioRemota");
        set("test",method); //no cal, pero així veiem com funciona
        try {
            for(;;) {
                try { Thread.sleep(Integer.MAX_VALUE); }
                catch (InterruptedException e) {}
            }
        } catch (Exception e) {
            System.out.println("Excepcio: " + e + "; " +
e.getMessage());
            e.printStackTrace();
        }
    }

    public static void main(String argv[])
    {
        throws RemoteException, NotBoundException, AlreadyBoundException

        Debug.debug = false; //dona informacio de la localitzacio dels
agents
        System.out.println("Creant agent "+argv[0]);
        Agent.init();
        robot a = new robot(argv[0]);
        System.out.println("Agent "+argv[0]+" creat\n");
        a.start();
    }
}

```

Acció remota dels robots: AccioRemota

L'acció que és cridada per l'agent món (*world*) i que tracta les seves crides és AccioRemota.java.

```

//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/AccioRemota.java

import kafka.Debug;
import kafka.agent.Action;
import java.util.Random;
import java.util.Date;
import java.rmi.AccessException;
import AgentOpInterficie;

public class AccioRemota extends Action {

    public static int x;
    public static int y;

    //procediment inicial
    protected void init() {
        System.out.print("Inicialitzem posicio. ");
        Random rand = new Random((long)System.currentTimeMillis());
    }
}

```

```

        int MAX = 200;
        x= Math.abs(rand.nextInt()%MAX);
        y= Math.abs(rand.nextInt()%MAX);
        System.out.println("Posicio inicial (x,y)=(\"+x+\",\"+y+\")\n");
    }

    //procediment cridat quan es fa una call()
    public Object start(Object arg[]) {

        if (arg == null || arg.length == 0) return null;

        String cadIn=(String)arg[0];
        String msgRet=null;
        if(arg[0].equals("inici")) {
            msgRet=x+\",\"+y;
        }
        else
        {
            String nomAgent = null;
            try {
                nomAgent = (String)self().get("name");
            } catch (Exception e) { }

            int xant=x;
            int yant=y;
            int strlen=cadIn.length();
            String numStr="";
            int num[]=new int[4];
            int j=0;
            for(int i=0; i<strlen; i++) {

                char c=cadIn.charAt(i);
                if(c==',') {
                    num[j]=Integer.parseInt(numStr);
                    numStr="";
                    j++;
                }
                if( (c!='\r') && (c!='\n') && (c!=' ') &&(c!=',')) {
                    numStr=numStr+c;
                }
            }
            num[j]=Integer.parseInt(numStr);

            x=num[0];
            y=num[1];
            if ((x!=xant)&&(y!=yant))
                System.out.println(nomAgent+"        brossa        tractada:
(x,y)=(\"+xant+\",\"+yant+\") a (x,y)=(\"+x+\",\"+y+\")");
            msgRet="Ok!";
        }
        return msgRet;
    }
}

```

12.3.2.2. Agent world

```
//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/world.java

import kafka.*;
import kafka.agent.*;
import java.rmi.*;
import java.util.Random;
import java.util.Date;
import TaulaRobots;
import AgentOpInterficie;

public class world extends Agent
{
    public TaulaRobots tr = new TaulaRobots();
    public static AgentOpInterficie op = new AgentOpInterficie("world");

    world()
        throws RemoteException, NotBoundException, AlreadyBoundException
    {
        super(null, System.getProperty("user.dir"));
        //tr = new TaulaRobots();
    }

    world(AgentOpInterficie intOp)
        throws RemoteException, NotBoundException, AlreadyBoundException
    {
        super(null, System.getProperty("user.dir"));
        op=intOp;
    }

    public void display(String cadena) {
        op.escriure(cadena);
        System.out.println(cadena);
    }

    public void run() {
        try {
            int rubx,ruby;
            String myname = (String)get("name");
            Object arg[] = new Object[1];

            //inicialitzem el sistema
            String llistaNoms[] = directory.list();
            arg[0]=new String("inici");

            display("Inicialitzant el sistema. Iteracio
["+i+"/"+MAX_ITERACIONS+"]");
            for(int i=0; i<llistaNoms.length; i++) {

                String nom=llistaNoms[i];
                if (nom.equals(myname)) continue;
                Remote ro = directory.lookup(nom);
                String ri = ro.getClass().getInterfaces()[0].getName();
                if(!ri.equals("kafka.agent.RemoteAgent"))
                    continue;
                RemoteAgent ra = (RemoteAgent)ro;

                String cadRet=(String)ra.call("test", arg);

                //s'ha d'extreure (x,y) de cadRet
                int strlen=cadRet.length();
                String numStr="";
                int num[]=new int[4];
                int j=0;
```

```

        for(int k=0; k<strlen; k++) {
            char c = cadRet.charAt(k);
            if(c==' ') {
                num[j]=Integer.parseInt(numStr);
                numStr="";
                j++;
            }
            if((c!='\r') && (c!='\n') && (c!=' ') && (c!=' ')) {
                numStr=numStr+c;
            }
        }
        num[j]=Integer.parseInt(numStr);

        int posx=num[0];
        int posy=num[1];
        //una vegada identificat el nou robot l'insertem en tr
(TaulaRobots)
        tr.insertar(posx,posy,nom);
        display("          <Nou          robot:          "+nom+"          a
(x,y)=("+posx+", "+posy+")>");
    }

    //Sistema inicialitzat, cal fer treballar els robots
    Random rand = new Random((long)System.currentTimeMillis());
    int MAX = 200;
    rubx = Math.abs(rand.nextInt()%MAX);
    ruby = Math.abs(rand.nextInt()%MAX);
    arg[0]=new String(rubx+", "+ruby);
    display("Brossa generada a (" +rubx+", "+ruby+)");

    int idx = tr.menorDistancia(rubx,ruby); //triem el mes proper

    int x1 = tr.consultaPosx(idx);
    int y1 = tr.consultaPosy(idx);
    String nomAgentTria = tr.consultaNom(idx);
    RemoteAgent          raTria =
(RemoteAgent)directory.search(nomAgentTria);
    display("#"+nomAgentTria+"          recull          brossa          i          passa          de
("+x1+", "+y1+") a (" +rubx+", "+ruby+)\n");

    String r=(String)raTria.call("test",arg);
    String cadRet=(String)raTria.call("test", arg);

    directory.unbind((String)get("name"));

    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static int MAX_ITERACIONS = 20;
public static int i;

public static void main(String argv[])
    throws RemoteException, NotBoundException, AlreadyBoundException
{

    Debug.debug = false;
    Agent.init();
    for (i=0; i<MAX_ITERACIONS; i++) {
        try { Thread.sleep(5000); }
        catch (InterruptedException e) {}
        world a = new world(op);
        a.start();
    }
}
}

```

12.3.2.3. Tauler de robots de l'agent món

```
//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/TaulaRobots.java
//

import kafka.*;
import kafka.agent.*;
import java.util.Date;

class TaulaRobots {

    int i=1;
    static final int INICI=1;
    static final int N=11;
    int x[] = new int[N];
    int y[] = new int[N];
    String nom[] = new String[N];

    public void insertar(int nx, int ny, String n) {
        x[i]=nx;
        y[i]=ny;
        nom[i]=n;
        i++;
    }

    public boolean hiHaAgents() {
        boolean ret=false;
        if(i>INICI) ret=true;
        return (boolean)ret;
    }

    public int consultaPosx(int indx) {
        return (int)x[indx];
    }

    public int consultaPosy(int indx) {
        return (int)y[indx];
    }

    public String consultaNom(int indx) {
        return (String)nom[indx];
    }

    public double calcularDist(int x0, int y0, int x1, int y1) {
        int a = x0-x1;
        int b = y0-y1;
        double d = (double)Math.sqrt(a*a+b*b);
        return d;
    }

    public int menorDistancia(int x1, int y1) {
        int indx=INICI;
        double dmin=999999;
        for(int j=INICI; j<i; j++) {
            double dtemp = calcularDist(x1,y1,x[j],y[j]);
            if(dtemp<dmin) { dmin=dtemp; indx=j; }
        }
        return (int)indx;
    }

    public void actualitzarPosicio(int indx, int nx, int ny) {
        x[indx]=nx;
        y[indx]=ny;
    }

    public void actualitzarPosicio(String id, int nx, int ny) {
        int j=INICI;
        boolean trobat=false;
    }
}
```

```

        while(!trobat && j<i) {
            if(nom[j].equals(id)) {
                actualitzarPosicio(j,nx,ny);
                trobat = true;
            }
            j++;
        }
    }

    public int consultaIndex(String id) {
        int indx=-1;
        int j=INICI;
        boolean trobat=false;
        while(!trobat && j<i) {
            if(nom[j].equals(id)) { indx=j; }
            j++;
        }
        return (int)indx;
    }
}

```

12.3.2.4. Interficie de world

```

//*****
// File: /mnt/d/pfc/kafka/cop_ex/rubbish/AgentOpInterficie.java

import java.io.*;
import java.util.*;
import java.lang.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;

class AgentOpInterficie extends Frame implements ActionListener,
WindowListener
{
    TextArea textArea;
    Panel p1;

    public TextArea getTextArea() { return textArea; }

    public AgentOpInterficie() {
        addWindowListener(this);
        generarPantalla("");
    }

    public AgentOpInterficie(String name) {
        addWindowListener(this);
        generarPantalla(name);
    }

    public void generarPantalla(String name) {
        setTitle(name);
        setSize(300,320);
        Panel p1 = new Panel();
        p1.setLayout(new GridBagLayout());

        Font f = new Font("Helvetica",Font.PLAIN,8);
        setFont(f);

        textArea = new TextArea(22,50);
        textArea.setEditable(false);
    }
}

```

```

        constrain(p1, new Label("Estat
Agent"), 0, 0, 1, 1, GridBagConstraints.NONE, GridBagConstraints.NORTHWEST, 0, 0);

        constrain(p1, textArea, 0, 1, 1, 3, GridBagConstraints.HORIZONTAL, GridBagConst
raints.NORTH, 1, 0);

        add("Center", p1);
        textArea.setText("");
    }

    public void escriureTitol(String nom) {
        setTitle(nom);
    }

    public void escriure(String cad) {
        textArea.append(cad+"\n");
        show();
    }

    public void constrain(Container con, Component component, int x, int y,
int width, int ht, int fill, int anchor, double wt_x, double wt_y)
    {
        GridBagConstraints c = new GridBagConstraints();
        c.gridx = x;
        c.gridy = y;
        c.gridwidth = width;
        c.gridheight = ht;
        c.weightx = wt_x;
        c.weighty = wt_y;
        ((GridBagLayout)con.getLayout()).setConstraints(component, c);
        con.add(component);
    }

    public void actionPerformed(ActionEvent event) {}

    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }

    public void windowClosed(WindowEvent event) {}
    public void windowDeiconified(WindowEvent event) {}
    public void windowIconified(WindowEvent event) {}
    public void windowActivated(WindowEvent event) {}
    public void windowDeactivated(WindowEvent event) {}
    public void windowOpened(WindowEvent event) {}
}

```

12.4. Aplicacions

S'han estudiat altres entorns de desenvolupament que oferien uns paquets de llibreries i l'usuari és qui les fa servir en una aplicació determinada, com per exemple JATLite o FAFMAS, però la diferència d'aquest entorn és la simplicitat. Ja s'ha vist que no proporciona gran quantitat de classes/mètodes, però les eines que proporciona són molt fàcils d'utilitzar i tenen diferents implementacions que poden satisfer nombroses aplicacions.

S'ha implementat un agent personal anomenat DUET. Els agents personals (PAs) són assignats a cada usuari d'una xarxa, i els ajuda a establir comunicacions amb altres usuaris de la xarxa, escollint els millors nodes i recursos.

Kafka va ser implementat el 1997 i d'ençà no s'ha desenvolupat cap més versió d'aquest software. Els dissenyadors senyalen l'existència (el 1997) de noves eines de desenvolupament de SMA però centrades en agents mòbils, com són *Aglets Workbench* i *Mole*. Actualment les eines de desenvolupament d'agents mòbils s'han multiplicat i Kafka ha passat a ser un referent dins aquest món. En la meua opinió crec que Kafka va impulsar el desenvolupament d'aquest tipus d'eines, però la metodologia emprada no era cap meravella i es va tendir a implementar altres tipus de sistemes.

Els dissenyadors de Kafka també senyalen que les llibreries encara no estan lliures d'errors, però nosaltres no hem detectat cap anomalia.

Es poden pensar moltes aplicacions pràctiques utilitzant aquest entorn de desenvolupament. L'únic problema pot estar en la complexitat que pot tenir una implementació segons quin sigui el domini.

Podem pensar en implementar aplicacions de gestió de serveis, com per exemple serveis de compra/venda, aplicacions de comerç electrònic, automatització d'oficines, o organització d'empreses.

Es pot pensar en implementacions d'agents mòbils en qualsevol domini. Es poden implementar aplicacions distribuïdes, implementar programes espies, o de recol·lecció d'informació de màquines remòtes, virus informàtics que viatgin a través de la xarxa, o agents dedicats a tasques específiques que calgui que viatgin per una xarxa electrònica.

En l'aplicació s'ha vist el funcionament d'una petita aplicació que simula la gestió de diversos robots, però aquests robots poden ser qualsevol cosa, com per exemple sensors d'una planta, i se'n pot gestionar el funcionament.

13. ESTUDI COMPARATIU

Les diferents eines que s'han estudiat en aquest treball, tot i ser diferents, poden veure's de manera global i comparant-ne diferents característiques.

Un punt en comú a tots els paquets de software és el llenguatge de desenvolupament. S'ha utilitzat Java. Aquest llenguatge és de recent aparició en el món de la informàtica, i incorpora moltes eines de desenvolupament i permet de crear-ne de noves.

Hi ha altres llenguatges de programació que també s'utilitzen en la programació de sistemes multiagent com són C, C++ o Lisp, però progressivament la seva utilització va minvant a favor de Java, i actualment la majoria d'eines que apareixen/actualitzen es fan en Java.

Dos elements característics de Java són els objectes (classes) i els fils d'execució (*threads*). Els objectes implementen mètodes i aquests poden ser cridats dinàmicament a mida que són necessaris; a banda de l'estructura de l'objecte cal un programa que vagi dirigint el funcionament. Els fils d'execució o *threads* poden entendre's com processos els quals tenen mètodes per tal de monitoritzar/programar la seva execució; tenen una autonomia que l'usuari programa per aconseguir un comportament determinat.

Segons s'esculli un mètode o un altre, això conduirà a un tipus de programació o altra.

Si ens trobem davant agents implementats com objectes, el comportament es realitza mitjançant crides de mètodes dels objectes. L'accés o crida de mètodes d'altres objectes pot realitzar-se de dues formes: a) Si els objectes o agents poden estar localitzats localment, la crida dels mètodes d'altres objectes és directa, b) Els objectes o agents poden estar distribuïts en diferents màquines, i cal la utilització d'un ANS que sàpiga la localització de tots els agents de la comunitat. El cas a) és l'utilitzat per Jack mentre que el mètode b) és l'utilitzat per Kafka.

La implementació sobre *threads* podríem dir que "dóna més joc", és a dir permet utilitzar unes altres eines i implementar altres tècniques.

Els *threads* proporcionen una autonomia innata als agents. Partint d'aquesta premissa, els agents han de comunicar-se. Per intercanviar missatges, es pot implementar un ACL definit per l'usuari i que tots els agents de la comunitat entenguin i, més enllà, es poden implementar tècniques de coordinació/negociació.

JATLite, JAFMAS i Zeus implementen com ACL l'estàndard KQML. Encara que hi ha altres ACLs disponibles, no hi ha masses eines de desenvolupament que els implementin, i en el mercat l'ACL més fàcil de trobar és KQML.

El protocol de transport implementat és TCP/IP o UDP/IP. Un estàndard àmpliament utilitzat en la transmissió de dades.

Actualment hi ha moltes eines de desenvolupament de SMA i l'usuari no ha de triar només allò que ofereix un sistema, sinó com ho ofereix. Per exemple, un sistema pot donar molt bones prestacions, però potser no ofereix cap tipus de facilitat a l'usuari com pot ser un entorn de desenvolupament amb una interfície home-computadora, llavors

l'usuari si ho ha d'implementar tot en la seva totalitat, pot requerir d'un temps d'aprenentatge molt més gran que en un altre sistema que, tot i no oferir tan bones prestacions, ofereix un entorn visual que li ajuda molt.

Arribar a un compromís entre prestacions i facilitats és feina de l'usuari que ha d'implementar un SMA. Potser Zeus no permet tantes aplicacions com JATLite o JAFMAS, però resulta molt més pràctic a l'hora de treballar i el seu aprenentatge és molt més ràpid.

Una característica important és la seguretat. Les eines estudiades en general no implementen facilitats de seguretat, ja sigui per establir les connexions entre els agents, o més enllà, en les transmissions de missatges. És per això, que aquestes eines permeten desenvolupar aplicacions de recerca o personals de l'usuari, però si es volen comercialitzar, cal incloure més prestacions per garantir aquest punt.

La majoria de productes són resultat de recerques per part d'Universitats, i normalment poden utilitzar-se per a finalitats de recerca, però també hi ha productes dissenyats per empreses privades que requereixen un pagament per a utilitzar-los.

En la meua opinió, no hi ha cap producte que sigui perfecte, per tant cal anar mirant quin paquet de software s'apropa més a les nostres necessitats, però sempre mirant que sigui de lliure distribució o d'alguna Universitat.

	JACK	JATLite	JAFMAS	ZEUS	KAFKA
Llenguatge de programació i versió	JAL però tradueix els fitxers font a fitxers Java (JDK 1.1.5)	Java (JDK 1.1)	Java (JDK 1.1)	Java (JDK 1.2)	Java (JDK 1.1)
Implementació dels agents	Objectes	Thread	Thread	Thread	Objectes
Comunicació entre els agents	ANS Centralitzat	ANS centralitzat i BroadCast.	Broadcast i Comunicació directa	ANS central	ANS central
Protocol de comunicació	UDP/IP	TCP/IP	TCP/IP i UDP	TCP/IP	Cap
Tipus de missatges	Missatges entre accions del sistema sense un format preestablert. Defineix un ACL propi.	Diferents tipus de missatges basats en la teoria dels actes de comunicació; es tenen performatives compatibles amb KQML.	Missatges basats en performatives compatibles amb l'estàndard KQML.	Missatges basats en performatives compatibles amb l'estàndard KQML.	Crides a mètodes remots amb paràmetres d'entrada i/o sortida.
Entorn de desenvolupament integrat	No amb reserves; es tenen una sèrie d'eines per compilar els fitxers i executar-los, però tot ho ha de fer l'usuari des de la línia de comandes.	No	No	Si, es proporcionen totes les eines per desenvolupar/executar qualsevol aplicació.	No
Proporciona eines de coordinació	Es té la coordinació intrínseca de la metodologia del sistema. La coordinació es proporciona alhora de definir els plans i els events que s'han d'anar duent a terme.	Es tenen les converses, però cal que el programador les dissenyi.	La coordinació es pot realitzar usant una màquina d'estats finits.	Si, es donen estratègies predefinides i l'usuari pot configurar. Es poden crear de noves segons les necessitats.	No. La coordinació/negociació entre els agents, s'han d'implementar segons es necessiti.
Eines de seguretat	No.	Es tenen les eines de seguretat proporcionades per Java, i a més es permet usar <i>username</i> i <i>password</i> per establir connexions (si es requereix)	Es tenen les eines de seguretat proporcionades per Java.	No.	Si, com el software està pensat per implementar agents mòbils, es té un paquet dedicat a eines de seguretat.
Tipus de software	JACK és un producte registrat. Cal comprar-lo per a fer-lo servir i segons la finalitat es poden obtenir descomptes especials (recerca, estudiants, etc.).	Tot el software es pot descarregar d'Internet sense problemes. Caldria demanar permís a l'autor per fer-ho servir, però segurament donaria el vistiplau per recerca/aplicació en una universitat.	Tot el software és de lliure distribució (<i>free copy</i>); es pot descarregar tot el software (llibries i documentació) d'Internet sense problemes. Podríem presentar-nos a l'autor identificant-nos i donant informació del seu ús.	El software pertany a BT i per utilitzar-lo cal demanar una llicència i segons la finalitat d'ús, cal pagar-ho.	El software pertany a Fujitsu Inc.. Es pot descarregar sense problemes d'Internet però caldria demanar permís, segons la utilització que se'n fes.



La tecnologia informàtica avança constantment. Aquest projecte té, ja des de la seva creació, una vida limitada. Internet proporciona un ventall molt gran de possibilitats i actualment hi ha un petit caos que cal solucionar.

Veient la llista d'eines que hom pot trobar per la xarxa, es fa difícil de treballar. Encara que és positiu que hagi molta oferta, per l'altra, es té molta feina en recuperar tota la informació, assimilar-la i treure'n conclusions per utilitzar-la.

No hi ha cap producte que s'hagi imposat, però es veu una tendència en el disseny d'eines de recent aparició. Aquestes noves versions o nous productes implementen potents interfícies per tal de facilitar la feina a l'usuari i deixen portes obertes per a què l'usuari implementi estratègies/opcions que li siguin òptimes pel seu problema.

Hi ha moltes pàgines web que parlen de SMA, però n'hi ha dues que crec són imprescindibles: la primera és <http://www.AgentBuilder.com/AgentTools.html> que glossa un seguit d'eines i dóna una petita descripció de cadascuna, i la segona és <http://www.multiagent.com>, imprescindible per estar al dia de totes les novetats que es van produint.

Esperem que aquest projecte hagi estat d'utilitat a tothom que el necessiti.

