Silviu RĂILEANU

Utilizarea platformei JADE pentru dezvoltarea aplicațiilor multi-agent

978-606-9608-39-5

editura POLITEHNICA PRESS

Utilizarea platformei JADE pentru dezvoltarea aplicațiilor multi-agent

Conf. dr. ing. Silviu RĂILEANU

Utilizarea platformei JADE pentru dezvoltarea aplicațiilor multi-agent

Editura POLITEHNICA PRESS
BUCUREȘTI, 2023

Adresă: Calea Griviţei, 132
10737, Sector 1, București
Telefon: 021.402.90.74

Referenți științifici:
Prof. univ. dr. ing. **Theodor BORANGIU**
Conf. dr. ing. **Florin ANTON**

ISBN: 978-606-9608-39-5

Cuprins

Capitolul 1: Tehnologii multi-agent

Cuprins

1. Agenți autonomi și sisteme multi-agent

Pentru un sistem de conducere descentralizat, Fig.1.1, este nevoie de un nou concept de abstractizare, concept care să țină cont de: a) noile caracteristici ale componentelor electronice, precum autonomie în procesul decizional și prezența capacităților de comunicare (*embedded systems*), b) noile impedimente precum complexitatea crescândă a sistemului în termen de comportament global și informații vehiculate, cât și de c) necesitățile de integrare a acestora în structurile superioare ale întreprinderii. O soluție pentru acestea ar fi utilizarea conceptelor de agent autonom/sistem multi-agent sau a conceptului de holon (sistem de fabricație holonic).

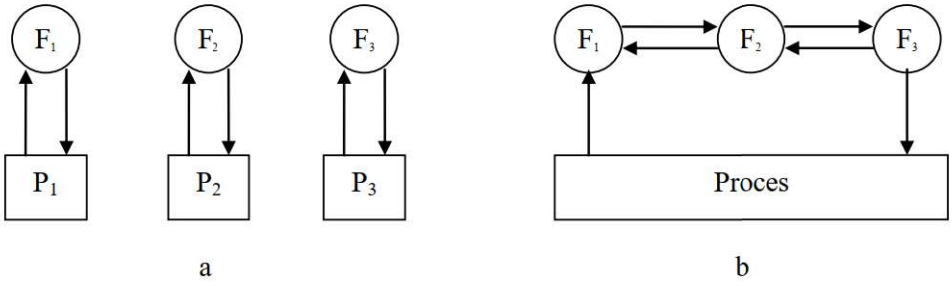

Fig.1.1 – Sistem de conducere: a. descentralizat şi b. distribuit

Conceptul de agent este folosit pentru modelarea unei entități autonome capabile să interacționeze cu mediul în care se află. Mai mulți agenți cu un scop comun care interacționează formează un sistem multi-agent (SMA). Conceptul este folosit atât pentru modelarea sistemelor descentralizate, compuse din mai multe entități autonome, cât şi pentru implementarea de aplicații descentralizate. Un exemplu îl constituie adaptarea acestor concepte în conducerea sistemelor de producţie unde a apărut noțiunea de holon: agent software care are cel mai adesea atașată o parte fizică (legătura fizic informațional).

Relativ la a doua utilizare a conceptului de sistem multi-agent (implementare de sisteme descentralizate de luare automată şi colaborativă a unei decizii) există o serie de unelte care facilitează dezvoltarea SMA. Principalele funcționalități ale acestora sunt: dezvoltarea de agenți generici şi posibilitatea instanțierii lor cu diferite caracteristici (identificator, proprietăți, date etc.), facilitarea interacțiunii dintre agenții componenți şi implementarea aspectelor decizionale sub formă de comportamente. Avantajele adiționale sunt: portabilitate, posibilitatea actualizării codului în timp real, toleranța la defecte, integrarea cu alte aplicații la nivel de servicii, integrarea cu baze de date pentru memorarea de informații importante. Dintre uneltele care permit dezvoltarea de sisteme descentralizate de luare a deciziei amintim: JADE (Java Agent Development Environment, *jade.tilab.com*), Jack (*http://www.aosgrp.com.au/*), Erlang (*http://www.erlang.org/*).

Conceptul de sistem multi-agent este derivat din domeniul Inteligenţă Artificială Distribuită (en.: *Distributed Artificial Intelligence* – DAI), fiind caracterizat de descentralizarea şi execuția paralelă a unor activități ce au la bază niște entități autonome denumite agenți. Termenul de agent se foloşeste în două cazuri: la modelarea sistemelor şi la implementarea aplicațiilor software distribuite. În cel de-al doilea caz, un agent este implementat întotdeauna printr-o aplicație, dar nu toate aplicațiile sunt agenți (ex.: Notepad este o aplicație pe când aplicația Word poate fi asociată unui agent, deoarece după citirea textului introdus poate propune corecturi pe baza unui dicționar selectat) (Wooldridge, 2002). Astfel, un agent reprezintă o entitate autonomă capabilă să perceapă mediul în care se află şi să acționeze asupra lui.

"An autonomous component, that represents physical or logical objects în the system, capable to act în order to achieve its goals, and being able to interact with other agents, when it doesn't possess knowledge and skills to reach alone its objectives." (Leitao, 2004)

Un sistem multi-agent este compus dintr-un set de agenți autonomi care interacționează în scopul realizării unui obiectiv comun.

Agenții autonomi reprezintă un nou mod de a analiza, proiecta şi implementa sisteme software complexe. Aceştia oferă o paletă largă de instrumente şi tehnici, care au potențialul de a îmbunătăți considerabil modul în care oamenii conceptualizează şi pun în aplicare mai multe tipuri de software. Agenții sunt din ce în ce mai utilizați într-o gamă largă de aplicații - variind de la sisteme relativ mici, cum ar fi filtre de e-mail personalizate până la sisteme complexe, cum ar fi controlul traficului aerian. La prima vedere, s-ar putea zice că astfel de sisteme au puțin în comun. Cu toate acestea, în ambele cazuri conceptul de abstractizare folosit este acela de agent.

Conceptul de agent a evoluat de-a lungul timpului din mai multe discipline: inteligenţă artificială, programarea orientată obiect şi proiectarea interfețelor om-mașină. Această moştenire, precum şi faptul că acest concept este folosit în varii domenii, precum: fabricaţie, logistică, planificarea producţiei etc. (Pechoucek, 2008), fac dificilă standardizarea unei definiții general acceptată. În literatura de specialitate s-au încercat diferite standardizări (Ferber, 1995; Jennings, 1998), însă nu există un consens general după cum reiese din (Franklin, 1996). Cu toate acestea, câteva caracteristici se regăsesc cu precădere, chiar dacă sub diferite forme, în majoritatea definițiilor: agenții autonomi trebuie să perceapă mediul în care se află, să fie pro-activi (să aibă comportamente orientate către atingerea unui scop: sunt independenți din punct de vedere decizional) şi să poată comunica cu alți agenți situați în același mediu. De asemenea, acest concept apare în special în domeniul aplicațiilor software şi, după cum se menționează în (Franklin, 1996), un agent este în general implementat printr-un program, singura diferență între un agent şi un program fiind aceea că un program este închis de utilizator (moare în timp).

Prin punerea la un loc a mai multor agenți autonomi cu același scop rezultă un sistem multi-agent, acesta reprezentând un concept foarte puternic în special în studierea sistemelor biologice, şi nu numai. Datorită progreselor din domeniul electronicii din ultimii ani s-a ajuns ca majoritatea utilajelor dintr-un sistem de fabricaţie (roboți industriali, sisteme transportoare, maşini CNC etc.) să fie controlate de un sistem pe bază de calculator, iar ansamblul lor să fie studiat ca un sistem multi-agent.

Acesta se caracterizează prin (Jennings, 1998):

- fiecare agent are o viziune incompletă a sistemului şi capabilități distincte;
- nu există un punct central de control;
- informațiile sunt descentralizate;
- execuția operaţiilor este asincronă.

Printre motivele care trezesc interes în studierea sistemelor multi-agent se găsesc: abilitatea de a oferi robustețe şi eficacitate în utilizare, abilitatea de a facilita interoperabilitatea cu sistemele moştenite şi, nu în ultimul rând, abilitatea de a rezolva probleme în care informațiile, experiența şi partea control sunt distribuite. În ciuda tuturor acestor avantaje, sistemele multi-agent trebuie să facă față unor provocări dificile (Jennings, 1998; Bond, 1988), precum:

1. Cum se formulează, descriu, descompun şi alocă probleme şi apoi cum se sintetizează rezultatele într-un grup de agenți inteligenți?
2. Cum comunică şi interacționează agenții? Ce limbaj de comunicare şi protocoale folosesc? Ce şi când comunică?
3. Cum se asigură faptul că agenții acționează coerent în luarea deciziilor, luând în calcul efectele nelocale ale luării unei decizii, precum şi evitarea unor interacțiuni dăunătoare?
4. Cum să se dea posibilitatea unor agenți individuali de a reprezenta şi a raționa despre acțiunile, planurile şi cunoştințele altor agenți, pentru a se putea coordona cu ei? Cum să raționeze despre starea procesului de coordonare (Ex.: inițiere sau terminare)?
5. Cum să se recunoască şi să se reconcilieze perspective diferite, precum şi intenții care intră în conflict, ale unor agenți care fac parte din același sistem şi care încearcă să își coordoneze acțiunile?
6. Cum să se balanseze calculul local cu comunicația? Mai general, cum să se administreze alocarea unui număr limitat de resurse?
7. Cum să se evite comportamente nedorite ale sistemului în caz de regim de operare haotic sau tranzitoriu?
8. Cum să se proiecteze şi să se dezvolte metodologii şi platforme pentru sisteme multi-agent practice?

Deoarece stabilirea unor definiții general valabile este un lucru greu de realizat, având în vedere diversitatea domeniilor şi aplicațiilor în care se folosesc agenții, comunitatea a încercat standardizarea altor aspecte care privesc acest concept. Un organism care a lucrat intens în domeniul agenților autonomi este FIPA (www.fipa.org), acesta aducând contribuții semnificative în direcția de cercetare ce privește comunicarea între agenți.

FIPA (*Foundation for Intelligent Physical Agents*) este o organizație care promovează tehnologiile multi-agent şi interoperabilitatea standardelor sale cu alte tehnologii. A fost înființată în 1996, pentru a produce specificații software pentru agenți eterogeni interacționând între ei şi sisteme multi-agent, iar din 2005 a devenit membru IEEE. În general, specificațiile produse au în vedere doar aspectul software al sistemelor, partea multi-agent, legătura dintre agentul software şi partea fizică fiind făcută prin sincronizare conform Fig.1.2.

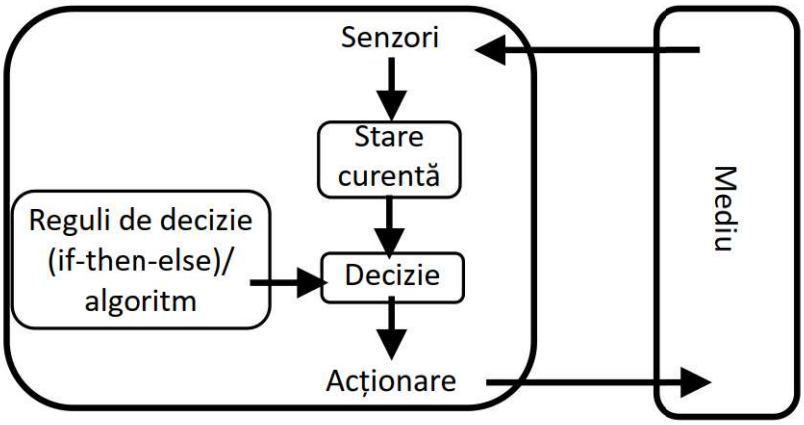

Fig.1.2 – Agent reflexiv simplu

După cum se menționează în literatura de specialitate (Duangsuwan, 2010), un sistem multi-agent este un sistem compus din mai multe entități de calcul identificate ca agenți. Originari din informatică, agenții au două caracteristici principale: i) operarea într-o manieră autonomă care se manifestă prin inteligența locală şi ii) capacitatea de a interacționa cu alți agenți – comunicare. În zilele noastre, în principal datorită progreselor din domeniul electronicii, autonomia a fost mult sporită, permițând agenților software să fie executați pe platforme mobile, dedicate, puternice (din punctul de vedere al capacităților de procesare) şi miniaturizate, cu rate reduse de consum de energie, crescându-le astfel mobilitatea (şi portabilitatea) şi în același timp şi modul în care percep mediul printr-o gamă largă de senzori şi conexiuni la rețea. Observațiile menționate anterior reflectă evoluția dispozitivelor încorporate care par a ne guverna viitorul, atât în ceea ce privește viața de zi cu zi (de ex.: oameni care interacționează cu software dedicat, cum ar fi asistentul personal), dar, mai important, în ceea ce privește domeniul industrial, așa cum este cazul sistemelor descentralizate

și eterogene. Astfel, a luat naștere un mediu interconectat, care se caracterizează prin descentralizarea puterii de calcul, aspect care merge mână în mână cu principiile SMA (Sisteme Multi-Agent). Prin urmare, formalismul SMA a fost adoptat pentru realizarea sistemelor de control descentralizat.

Cu originile înrădăcinate în domeniul inteligenței artificiale (Wooldridge, 2004), formalismul multi-agent a câștigat mai multă atenție în ultimii ani, așa cum se poate observa din numărul de publicații pe această temă (Fig.1.3). Având în vedere numele său particular, care nu are suprapuneri cu alte domenii nerelevante, o simplă căutare folosind cuvântul cheie „multi-agent“ într-o bază de date bibliografică standard (www.sciencedirect.com) care conține informații din reviste academice, lucrări ale conferințelor și alte documente din diverse discipline academice, relevă un interes sporit pentru acest domeniu (v. Fig.1.3). Chiar dacă formalismul multi-agent este folosit pentru a rezolva o mare varietate de probleme, două domenii principale de utilizare pot fi identificate printr-o analiză amănunțită a literaturii. Aceste domenii sunt: i) cercetare/teoretică, pentru modelarea, simularea și optimizarea sistemelor care sunt în mod inerent descentralizate (cum ar fi, sisteme de control compuse din resurse diferite și separate geografic, lanțuri de aprovizionare, distribuție de utilități – gaze, energie sau chiar populații de animale) și ii) practice, pentru a sprijini dezvoltarea eficientă a sistemelor descentralizate (de control) folosind tehnologii bazate pe agenți, cum ar fi cadre de dezvoltare multi-agent (Bellifemine et al., 2007), instrumente de lucru (Marcon et al., 2017; MaDKit, 2022) și chiar metodologii, deoarece unele limbaje de programare chiar dacă sunt utilizate pentru implementarea sistemelor distribuite (Krzywicki et al., 2015) nu sunt în mod inerent etichetate ca multi-agent.

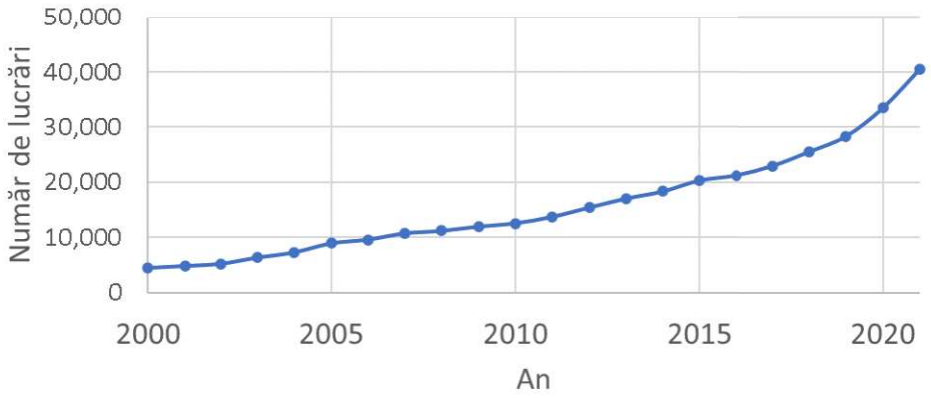

Fig.1.3 Evoluția numărului de articole pe tema „multi-agent” în ultimii 20 de ani

În acest sens, obiectivele acestui capitol introductiv sunt de a analiza lucrările relevante utilizând formalismul SMA. Sunt luate în considerare articole științifice, proiecte/implementări și platforme de dezvoltare multi-agent. Astfel, se va face o sinteză a principalelor domenii ce folosesc SMA cu accent pe domeniul producției, atât ca element de sine stătător, cât și ca parte a lanțului de valoare adăugată. În capitolele ce urmează vor fi prezentate: i) aplicațiile sistemelor multi-agent pentru modelarea, simularea și automatizarea proceselor, ii) principalele platforme de dezvoltare a agenților și iii) utilizarea SMA în conducerea fabricației.

2. Sisteme multi-agent pentru automatizarea proceselor, modelare și simulare

Automatizarea și simularea proceselor au fost recunoscute de mediul academic și de industrie ca domenii vitale de cercetare, cu multe programe de cercetare și proiecte industriale inițiate pentru investigare. Starea actuală a unor astfel de proiecte se caracterizează printr-o creștere a complexității datorită numărului mare de entități implicate și, în consecință, a interacțiunilor dintre acestea. Acesta este motivul pentru care formalismul SMA a fost folosit pentru astfel de sisteme. O analiză amănunțită a cercetării actuale (Herrera et al., 2020) a făcut diferența între: a) sisteme multi-agent (identificate simplu ca SMA), un concept utilizat pentru împărțirea muncii între multiple noduri care cooperează și iau decizii, utilizate în mod tradițional pentru implementarea sistemelor de conducere și monitorizare descentralizate și b) simulare și modelare multi-agent (*Agent Based Modelling and Simul*ation – ABMS) (Lane, 2014), un concept care se concentrează pe analiza comportamentului individual și colectiv al agenților într-un mediu simulat. În funcție de utilizarea sa, formalismul SMA poate fi aplicat la o mare varietate de domenii, cum ar fi: automatizarea proceselor de producție din fabrici (Leitão et al., 2017), sisteme de producere și distribuție a energiei (Woltmann et al., 2020), automatizarea clădirilor (Leitão et al., 2017; IEEE, 2020), procesele de afaceri (Jamison, 2017), rețelele de comunicații (Enrique, 2008), în principal domeniul ingineriei sistemelor (Herrera, 2020) în care conceptele și cadrele sunt utilizate pentru a dezvolta sisteme de lucru (de exemplu, sisteme de control descentralizate). ABMS este folosit cu precădere pentru a analiza comportamente emergente, procese de autoorganizare și pentru evaluarea de teorii, arhitecturi SMA și protocoale de interacțiune în domenii precum chimie (Mostafa et al., 2018), biologie (Ginovart et al., 2012), finanțe (Damaceanu, 2008; Mohamed et al., 2018), transport (He et al., 2021), dar și pentru optimizarea proceselor (prin simulări) (Muravev et al., 2021).

În cele ce urmează vor fi prezentate aplicații reprezentative ce folosesc formalismul SMA și unelte de dezvoltare necesare rezolvării problemelor complexe de conducere și optimizare în domeniile prezentate anterior.

a. Conducerea producției și automatizarea proceselor de fabricație

Descentralizate prin definiție, SMA, în special soluțiile de dezvoltare de aplicații multi-agent, sunt folosite în ingineria sistemelor pentru a implementa partea de conducere a sistemelor ale căror modele seamănă cu rețele complexe (Herrera et al., 2020). Unul dintre principalele domenii în care s-au obținut rezultate bune este automatizarea fabricii, așa cum este raportat în (Kruger et al., 2013). Aceste rezultate au promovat ceea ce autorii au numit agenți industriali (IEEE, 2020). Autorii propun o soluție pentru a integra agenți software cu dispozitive hardware fizice și un set de practici de interfață. În acest sens, se diferențiază două niveluri decizionale (Fig.1.4): a) conducere de nivel jos, asociată cu controlul în timp real și dispozitive fizice (de obicei încorporate) precum PLC-uri și calculatoare industriale, capabile de conexiune electrică directă la procesul controlat și b) conducere de nivel înalt care implementează sarcini strategice de luare a deciziilor, cum ar fi optimizarea, integrarea/comunicarea, configurarea etc.

Figura 1.4 ilustrează procesul de asociere a unui agent software unei componente fizice (Răileanu et al., 2014). Astfel, fiecare entitate de conducere are asociat un agent software responsabil cu procesul de luare a deciziilor și cu comunicarea printr-un mediu standard. Acești agenți vor fi văzuți ca obiecte de automatizare – abstracție de dispozitive mecanice cu inteligență încapsulată (Obitko et al., 2002) permițând astfel reutilizarea componentelor.

De obicei, această conducere de nivel înalt este implementată folosind platforme SMA, dar pentru că sunt necesare capacități de procesare ridicate și deoarece există limitări în ceea ce privește natura programelor executate pe dispozitivele de control de nivel jos (de exemplu: un PLC nu poate executa un agent Java, precum JADE), stratul de conducere la nivel înalt nu poate funcționa în timp real. Recomandările sunt astfel calculate la nivelul SMA și sunt sincronizate cu nivelul de control scăzut în moduri diferite, așa cum este prezentat în figura 1.5. Acest concept de interfață a evoluat de la agenți software asociați produselor inteligente (Meyer et al., 2009) la agenți industriali asociați fiecărei entități implicate în procesul de fabricație. În acest sens, axa inițială locală și la distanță, care descrie locația informațiilor (agent), a fost acum completată cu o altă axă, asociată procesului de implementare, ce detaliază modul în care se realizează interacțiunea (strâns – ex.: apel RPC și cuplat liber).

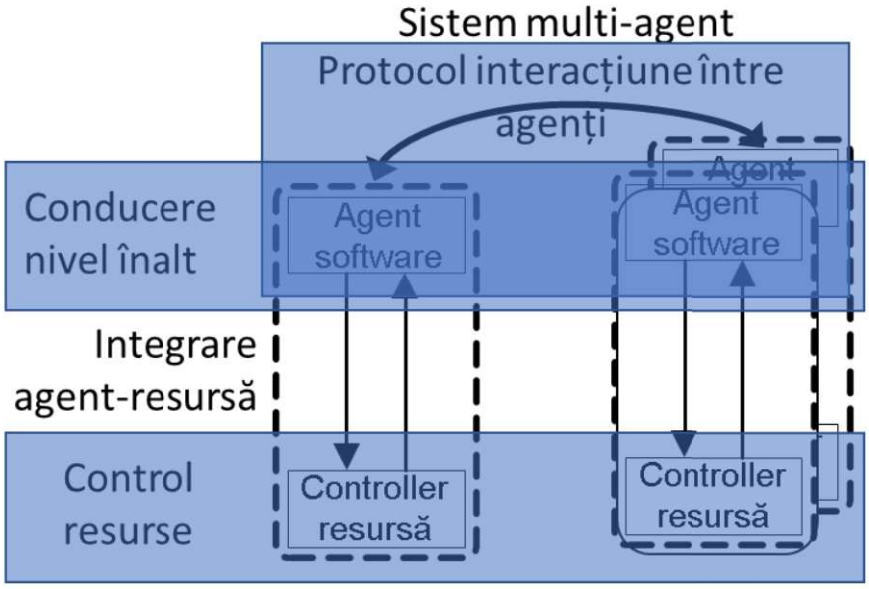

Fig.1.4 Resurse fizice cu agenți asociați (Răileanu et al., 2014) și interacțiunea dintre agenții software și dispozitivele de conducere a procesului automatizat (Leitão et al., 2016)

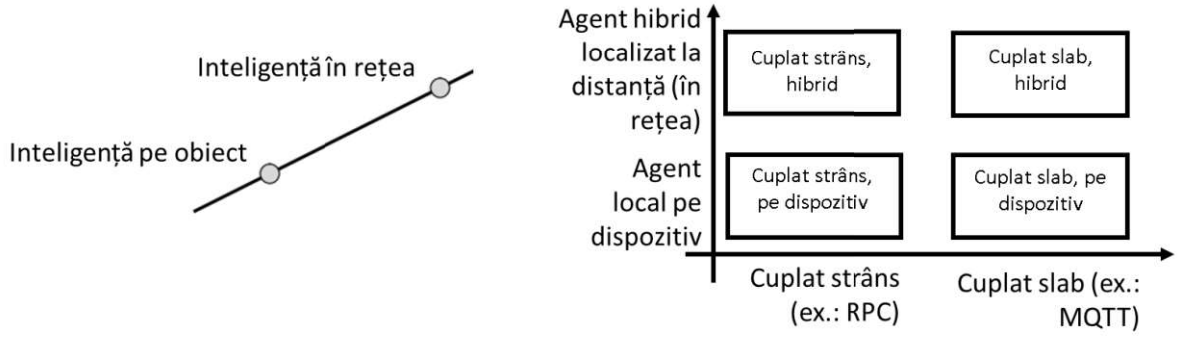

Fig.1.5 Localizare diferită a modurilor de luare a deciziilor și de interacțiune cu agentul software (Leitão et al., 2017; Leitão et al., 2016; Meyer et al., 2009)

b. Sisteme de distribuție a energiei electrice

Tendința actuală în generarea de energie se concentrează pe tranziția de la combustibili fosili la energia regenerabilă, fiind caracterizată prin generare descentralizată și luare a deciziilor inteligente, atât din partea consumatorului, cât și a furnizorului. În acest sens, formalismul SMA, dar și cadrele de dezvoltare SMA sunt folosite pentru a modela astfel de sisteme de distribuție a energiei și, în consecință, pentru a le implementa partea decizională. Un astfel de exemplu este raportat în (Woltmann et al., 2020), unde autorii prezintă problema cu aspectele sale: SMA/descentralizate, cerințele importante ale pieței de energie și, în final, o implementare efectivă folosind un cadru comun SMA – JADE. Ca și în domeniul producției, sistemele energetice împart problema de conducere în două straturi: a) un strat cibernetic, care

modelează o centrală electrică virtuală (VPP) responsabilă cu colectarea ofertelor și solicitărilor de la furnizorii și consumatorii agenți, implementate printr-un agent VPP (*Virtual Power Plant*) și b) un strat fizic, responsabil cu implementarea agenților în medii de rulare, etichetat aici ca unități tehnice (TU - *Technical Unit*). Aceste TU sunt împărțite în: b1) dispozitive de automatizare, cum ar fi PLC-uri sau plăci System-on-a-Chip cu interfețe Ethernet, care implementează funcția de gateway pentru a comunica cu sistemele moderne și vechi de magistrală de câmp capabile să controleze direct infrastructura electrică și b2) hardware, cum ar fi PC-uri industriale sau gateway-uri IoT responsabile pentru rularea de agenți TU de nivel scăzut. În acest sens, arhitectura propusă se bazează pe protocoale de interacțiune FIPA (www.fipa.org), unde agenții VPP reprezintă inițiatorii, în timp ce agenții TU acționează ca participanți.

Articolul în cauză prezintă o soluție (Fig. 1.6) care reduce decalajul dintre cercetarea SMA și aplicarea acesteia în industrie, energie și sisteme energetice în acest caz. Implementarea validează, de asemenea, că utilizarea unui cadru SMA dedicat (JADE), a protocoalelor standard de interacțiune (FIPA), a infrastructurilor hardware dedicate (PLC-uri, gateway-uri IoT, PC-uri industriale) precum și a platformelor software (OPC, servicii web, MODBUS TCP, NodeRed) este posibil să răspundă cerințelor pieței, cum ar fi operarea în timp real (1 secundă pentru procesele de luare a deciziilor), toleranță la defecte/redundanță, întrerupere a infrastructurii de comunicație/Internet, precum și aspectelor ce țin de securitate prin utilizarea rețelelor private.

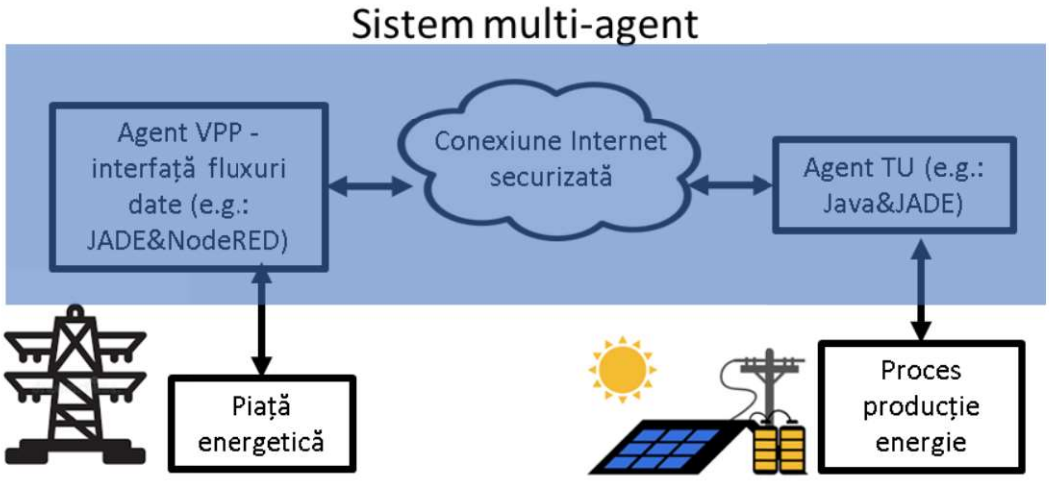

Fig.1.6 Exemplu de implementare a unui sistem de control al energiei inspirat de SMA (Woltmann et al., 2020)

c. Automatizarea clădirilor

O clădire inteligentă este o clădire echipată cu dispozitive încorporate pentru a implementa adaptarea automată la condițiile de mediu în continuă schimbare și pentru a oferi condiții de viață confortabile pentru ocupanții săi. Obiectivele principale ale unui sistem de automatizare a clădirilor sunt: optimizarea consumului de energie, reducerea costurilor de operare (automatizare control acces), securitate, control și operare de la distanță și îmbunătățirea utilizării echipamentelor. În mod tradițional realizate într-un mod centralizat, în prezent, datorită complexității crescute a unor astfel de sisteme, tehnologiile SMA sunt folosite pentru a asigura funcționarea descentralizată. Ca și în cazul secțiunii c) (Sisteme de distribuție a energiei), SMA este folosit aici pentru a automatiza minimizarea consumului de energie.

În acest sens, autorii referinței (Duangsuwan et al., 2010) propun un sistem multi-agent al cărui scop este de a combina controlul confortului cu o strategie de economisire a energiei. SMA-ul rezultat este compus din patru tipuri de agenți, fiecare implementând un aspect specific al sistemului de automatizare a clădirii: i) *agentul ocupant* este un agent reprezentativ uman care monitorizează și se adaptează la activitățile utilizatorului, învață stilurile și preferințele utilizatorului; ii) *agentul de zonă* este un agent responsabil cu controlul unei anumite zone, luând în considerare datele senzorilor și acționând ca un negociator între el însuși și agentul ocupant; iii) *agentul manager* este o interfață directă cu sistemul de management al clădirii; iv) *agentul de control al mediului* monitorizează și controlează diferiți parametri de mediu în fiecare zonă.

Sistemul a fost proiectat pentru două tipuri de scenarii: un singur ocupant, în care decizia se ia prin reconcilierea între ocupant și agenții de mediu și scenariul cu mai mulți ocupanți, în care decizia se ia prin negociere între agenții ocupanților folosind un sistem de tip "blackboard".

d. Rețele complexe

Cercetarea în zilele noastre are trei provocări majore: i) distanțe mari (de exemplu, atingerea altor planete), ii) distanțe mici (de exemplu, manipularea microorganismelor) și iii) gestiunea complexității (de exemplu: tratarea sistemelor interconectate, controlul comportamentului lor emergent și asigurarea coerenței/convergenței). Modelând aceste sisteme interconectate ca rețele complexe, autorii (Herrera et al., 2020) arată cum formalismul SMA poate fi aplicat pentru a controla și optimiza astfel de sisteme. O mare majoritate a

sistemelor distribuite pot fi modelate ca grafuri în care nodurile reprezintă entitățile care le compun, iar muchiile reprezintă relațiile/comunicațiile/interacțiunile dintre ele. În practică, aceste modele au topologii neregulate pentru a reprezenta mai bine sistemele din lumea reală. Acest tip de model se numește rețea complexă. Rețelele de calculatoare, Internetul, rețelele sociale (Enrique et al., 2008), rețelele de utilități (Woltmann et al., 2020; Obitko et al., 2002), rețelele de transport (Odell, 2011; Lu et al., 2008), sistemele chimice, biologice și biochimice (Ginovart et al., 2012) sunt câteva exemple de sisteme din lumea reală care pot fi modelate ca rețele complexe. Formalismul SMA poate fi aplicat pentru a analiza diferite scenarii. În acest sens, autorii separă cadrele SMA utilizate pentru implementarea sistemelor descentralizate, de modelarea și simularea bazată pe agenți (ABMS), utilizate ca instrument pentru analiza seturilor de entități individuale modelate ca agenți. Articolul de cercetare analizat în (Herrera et al., 2020) prezintă un set de aplicații reprezentative ale rețelelor complexe și ale controlului bazat pe agenți, care sunt utilizate pentru implementarea unor procese importante în diferite domenii ale Ingineriei Sistemelor, cum ar fi: producție și lanțuri de aprovizionare (programare, procesare simultană a comenzilor, asigurarea calității, personalizarea în timp real și întreținerea), rețeaua de energie electrică (sistemele multi-agent care utilizează contoare inteligente implementează controlul distribuit pentru sistemele de energie, oferind unele dintre funcțiile de supraveghere automată de tip ierarhic sau central care cresc capacitatea sistemului, fiabilitatea și eficiența economică), sisteme de transport (sistemele ciber-fizice compuse din agenți asociați vehiculelor și nodurilor de transport sunt dispuse într-o arhitectură de control ierarhică în care deciziile strategice sunt luate la nivelul rețelei și deciziile operaționale, influențate de stratul superior, sunt implementate la nivelul vehiculului), distribuție de utilități (sistemele bazate pe agenți se potrivesc bine cu ubicuitatea din zilele noastre a senzorilor utilizați pentru a opera surse individuale de gaz/apă și puncte de cerere).

Pe baza acestor analize de sisteme descentralizate se poate trage un set de concluzii parțiale: a) SMA pot reprezenta exemple mai realiste de cazuri reale, b) SMA va juca un rol esențial oferind rețelelor complexe inteligență pentru conducerea fluxurilor, evoluție și securitate, c) validitatea modelelor bazate pe agenți este dată de datele reale achiziționate printr-o rețea largă de senzori, d) inteligența artificială prin învățarea automată va fi folosită pentru a dezvolta noi modele și comportamente de agenți, e) termenul SMA evoluează în termenul de sistem ciber-fizic (*Cyber-Physical System*) datorită legăturii mai strânse dintre straturile informaționale și fizice, f) SMA poate fi folosit în tehnologia blockchain.

e. Analiza și securitatea rețelelor

Folosite în dezvoltarea sistemelor descentralizate, SMA funcționează întotdeauna într-un mediu digital, implementat ca o rețea locală de calculatoare, Internet sau, mai frecvent, dispozitive încorporate (*embedded devices*). Cu aplicații care acoperă o multitudine de domenii, unele dintre ele cu cerințe critice, cum ar fi distribuția de energie electrică (Woltmann et al., 2020), securitatea și analiza traficului devin factori cheie în dezvoltarea unui astfel de sistem de conducere descentralizat, care ar trebui să fie fiabil și rezistent în fața perturbațiilor și a evenimentelor cibernetice negative. În acest sens, autorii (Choi et al., 2020) propun un sistem de automatizare a distribuției (*distribution automation system* – DAS) implementat folosind un SMA, acesta din urmă utilizând algoritmi de detectare și atenuare care pot identifica anomaliile, activitățile anormale și operațiunile neobișnuite ale sistemului DAS. Astfel, funcționarea este împărțită între un set de agenți stratificați, responsabili cu funcționarea centralizată, întreruperea și controlul unităților terminale. Formalismul SMA din acest raport este folosit pentru a modela interacțiunile dintre agenții componenți care se monitorizează pentru a detecta atacurile de tip Denial-of-Service (DoS).

Într-un alt studiu (Álvaro et al., 2009), formalismul SMA este folosit pentru a implementa un sistem de detectare a intruziunilor (*Intrusion Detection System* – IDS), în care agenții software implementează funcții de monitorizare a securității direct la elementul gazdă. Arhitectura prezentată este împărțită în module implementate ca agenți software: agenți reflex (care detectează mediul și în plus realizează jurnalele de operare și generează alarme), agenți de analiză (responsabili cu analiza atacului pe baza datelor capturate de agenții reflex) și agenți de decizie (care acționează ca agenți bazați pe obiective pentru a lua deciziile adecvate). În plus față de arhitecturile IDS fixe, autorii celor de mai sus studiază utilizarea agenților mobili (agenți care migrează de-a lungul diferitelor gazde din rețea pentru a le monitoriza) pentru a detecta evenimentele care pot avea legătură cu intruziuni. Principalele avantaje pe care le oferă agenții mobili sunt: a) mobilitatea (prin relocarea agenților de analiză și decizional mai aproape de sursă scade latența răspunsului), b) robustețea și c) toleranța la erori (datorită replicării agenților). În ecosistemul digital interconectat de astăzi, care reprezintă mediul de operare pentru SMA, apariția atacurilor cibernetice este o preocupare importantă pentru funcționarea procesului sistemului. Din punctul de vedere al controlului procesului, atacurile cibernetice interferează de obicei cu sistemele de tip SCADA (Herrera et al., 2020) (*Supervisory Control and Data Acquisition*), afectând agenții software conectați direct la Internet.

f. Automatizarea proceselor de afaceri (agenți de tip RPA)

Deși implementat ca soluție de luare a deciziilor în mod distribuit (în rețea), SMA poate fi folosit la nivelul entităților sale componente – agenții – care se caracterizează prin inteligență locală și interacțiune bidirecțională cu mediul (percepție și acțiune). În acest sens, agenții software pot fi folosiți pentru automatizarea sarcinilor aferente proceselor de afaceri (de obicei legate de documente), cum ar fi: completarea automată a documentelor digitale, digitalizarea documentelor scrise folosind OCR etc. Procesele menționate anterior sunt realizate de agenți software în timp ce sunt subordonate unui sistem centralizat. Așa a apărut termenul *Robotic Process Automation* (RPA): utilizarea de software care încorporează tehnologii precum inteligența artificială (*artificial intelligence* – AI) și învățarea automată (*machine learning* – ML) pentru a automatiza sarcini de rutină de mare volum, care sunt sensibile la erorile umane (Jaminson, 2017). Agenții reprezintă aplicații software care captează mediul computerului (nivelul prezentare așa cum este văzut/perceput de operatorul uman) și operează într-un timp accelerat permițând procesarea unor volume mari de date. Prin integrarea și utilizarea sistemelor proprietare, agenții RPA pot funcționa într-un mod aproape automat. O arhitectură clasică (Fig1.7) este compusă din mai mulți agenți RPA autonomi interconectați printr-o bază de date comună cu alți agenți RPA situați pe același nivel decizional și cu agenți de nivel superior (cum ar fi MES în cazul producției). Agenții RPA sunt supuși aceleiași monitorizări, analize și control în vederea raportării ca și omologii lor umani. Deși proiectați pentru automatizarea sarcinilor aferente domeniului afacerilor, agenții RPA sunt folosiți și în procesele industriale (Abraham et al., 2021) pentru verificarea automată a stării, configurarea echipamentelor, urmărirea materialelor, detectarea alarmelor și colectarea datelor, mai ales în situațiile în care integrarea totală nu este posibilă, iar dezvoltatorul folosește nivelul de prezentare pentru automatizarea sarcinilor. Dacă integrarea totală este posibilă, agenții RPA sunt percepuți ca agenți industriali (IEEE, 2020) și interacționează prin protocoale standard de comunicație cu controlul de nivel scăzut (de exemplu, OPC, MODBUS a.o.).

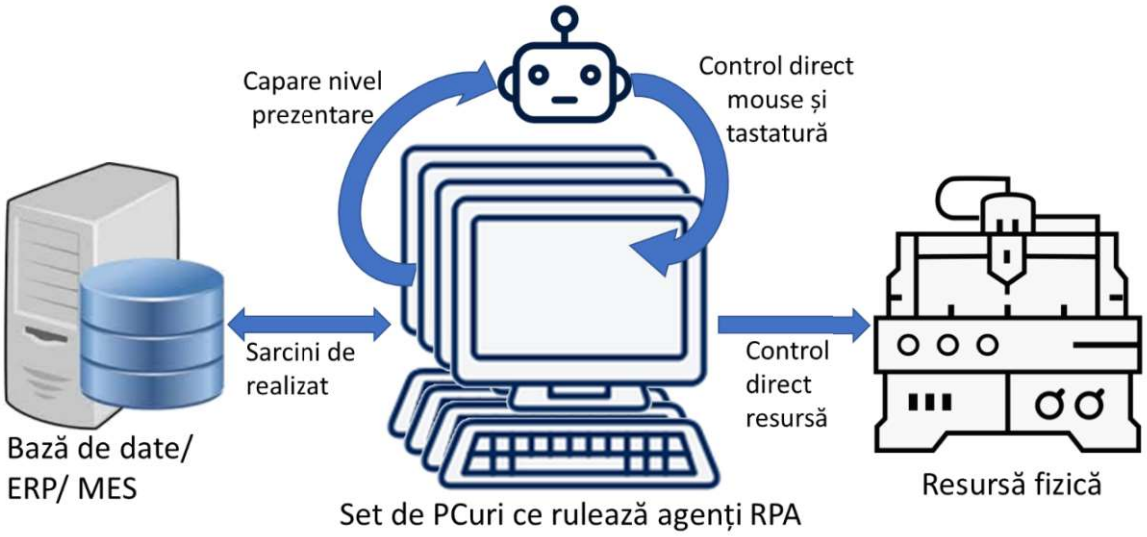

Fig.1.7. Modul de operare al unui agent RPA

g. Aplicații ale SMA în modelarea și simularea proceselor

Mai numeroase, aplicațiile în care este utilizat conceptul ABMS sunt potrivite în general pentru științe cognitive, interacțiuni sociale complexe (Herrera et al., 2020) și studiul sistemelor biologice sau chimice în care componentele individuale (microbi, molecule, bacterii, atomi) sunt modelate ca agenți. În acest sens, rezultă un SMA în care agenții pot interacționa într-un timp accelerat (pentru a obține rezultate mai rapid decât efectuarea experimentului fizic); este de asemenea posibilă modelarea și simularea evoluției sistemelor teoretice, greu de replicat, precum bioreactoarele (Ginovart et al., 2012)) sau potențial periculoase (sisteme microbiene (Taherian et al., 2018; Ginovart et al., 2012)). După cum s-a menționat la începutul capitolului, fundamentele acestei ramuri a sistemelor multi-agent, și anume modelarea și simularea bazată pe agenți (ABMS), diferă de formalismul standard SMA. În acest sens, au fost dezvoltate diverse platforme de modelare și simulare bazate pe agenți pentru a face față unor astfel de interacțiuni complexe, cum ar fi NetLogo (Netlogo, 2022; Marcon et al., 2017), MaDKit (MaDKit, 2022;Gutknecht, 2000) și Repast (Repast, 2022; North et al., 2013). În literatură există două abordări de modelare a unui sistem viu (Ginovart et al., 2012): a) abordarea bazată pe populație, o abordare de sus în jos care se bazează pe legile și modelarea matematică și b) modelarea bazată pe indivizi, o abordare discretă, de jos în sus, care se bazează pe modelarea entităților individuale și a interacțiunilor între acestea; aceste interacțiuni sunt atât cu entitățile echivalente cât și cu mediul. Cele două abordări nu se exclud reciproc, dar sunt mai degrabă complementare în sensul că modelarea matematică

(caracteristică abordării bazate pe populație) poate fi utilizată pentru a modela interacțiunile individuale. Astfel, prin aplicarea formalismului SMA, rezultă modele matematice realiste care pot fi utilizate pentru a simula cu acuratețe procese precum: bioreactoare, uzine chimice sau procese biologice (de exemplu: model pradă-prădător). Un exemplu de model pentru un proces biologic este dat în figura 1.8, unde animalele (lupii și oile) sunt modelate ca agenți mobili, în timp ce mediul este modelat ca agent staționar.

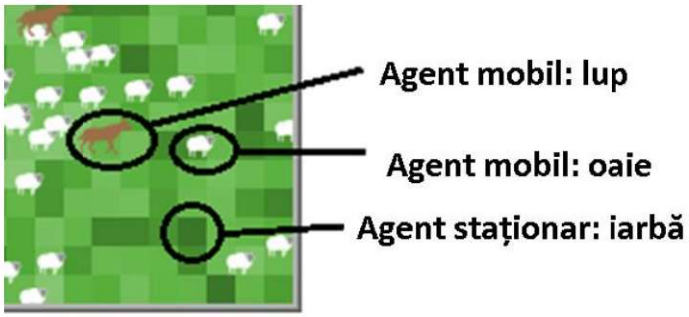

Fig.1.8 Transformarea unui proces biologic într-un model individual bazat pe agenți (NetLogo, 2022)

3. Platforme multi-agent

O platformă de dezvoltare de aplicații multi-agent este un sistem software utilizat de agenți și de mediu pentru a executa operațiuni. În acest ecosistem software, agenții operează pentru a-și atinge obiectivele (Leszczyna, 2008). Platforma multi-agent poate oferi funcționalități necesare dezvoltării de agenți, aplicații agent sau simulări bazate pe agenți. În acest sens, există două tipuri de platforme multi-agent: a) platforme utilizate pentru programarea orientată pe agenți (*agent oriented programming* – AOP) (Odell, 2011) și b) platforme pentru modelarea bazată pe agenți (*agent based modelling* – ABM) (Odell, 2011). Din punct de vedere istoric, AOP este o evoluție a conceptului OOP unde unitatea de bază a paradigmei de programare este agentul. AOP modelează și implementează un proiect ca o colecție de componente numite agenți care se caracterizează prin autonomie, proactivitate și capacitatea de a comunica. În cazul AOP este corect să vorbim despre proiecte, nu doar despre aplicații, deoarece un SMA este compus din mai mulți agenți software, fiecare dintre ei reprezentând o aplicație diferită, rulând de obicei pe mașini diferite. O distincție în acest sens poate fi făcută între sistemele middleware și cele orientate spre raționament (Braubach et al., 2005). Sistemele orientate spre middleware se concentrează pe inteligența socială a agenților (comunicare) și modul în care aceasta poate fi standardizată atât la nivel de mesaj (structură, gramatică), cât și la nivel de protocol de interacțiune (secvență de mesaje). Cea mai citată

referință pentru interoperabilitate, care este utilizată în dezvoltarea majorității platformelor AOP este *Foundation of Intelligent and Physical Agents* (FIPA, www.fipa.org). Sistemele orientate spre raționament se concentrează pe inteligența locală și luarea deciziilor, folosind în principal modelul de dezvoltare software BDI (*Belief Desire Intent*).

Din bibliografia analizată, platforma multi-agent JADE (Bellifemine et al., 2007) este cel mai utilizat element pentru dezvoltarea de aplicații multi-agent. Din punct de vedere tehnic, orice limbaj de programare multiplatformă (*cross-platform*) care permite implementarea de soluții de comunicare în rețea poate fi folosit pentru a dezvolta proiecte descentralizate care respectă cerințele SMA (autonomie, capacitate de comunicare și luare a deciziilor). Pe baza unui studiu amănunțit al platformelor AOP disponibile de-a lungul anilor (Kalliopi et al., 2015; Leitão et al., 2016) s-a observat că cele mai multe dintre ele tind să fie scoase din uz foarte repede, așa cum este cazul majorității produselor din domeniul IT (Leszczyna, 2008). În consecință, în această prezentare vom menționa doar cele mai utilizate platforme, așa cum sunt prezentate în tabelul 1.1. O comparație exhaustivă a diferitelor alternative poate fi găsită în (Kalliopi et al., 2015) cu unele completări în (Leitão et al., 2013).

Tabelul 1.1 Comparație între diferitele platforme de tip AOP

Platformă AOP	Cross platform	Ultima versiune	Limbaj programare	Observații
JADE (Bellifemine et al., 2007)	Da	JADE 4.5, 2017	Java	Platformă de tip open-source pentru dezvoltarea de aplicații agent peer-to-peer, https://jade.tilab.com
WADE(Federico et al., 2014; WADE, 2022)	Da	WADE 3.6.0, 2017	Java	Platformă open-source bazată pe JADE pentru automatizarea fluxurilor de lucru în aplicații multi-agent
JADEX (Brabach et al., 2006;, JADEX, 2022)	Da	Jadex 4, 2021	Java și XML	Implementarea unei infrastructuri BDI pentru agenții JADE
JACK (Winikoff, 2005; JACK, 2022)	Da	JACK 5.6, 2015	Extensie Java	Platformă multi-agent comercială cu un limbaj proprietar pentru modelarea capacității decizionale a agenților. Folosește modelul software BDI.
Erlang (Krzywicki et al., 2015; Kruger et al., 2019)	Da	Erlang/ OTP25, 2022	Erlang	Limbaj de programare folosit pentru a construi sisteme soft descentralizate în timp real, masiv scalabile, cu cerințe de disponibilitate ridicată.

Spre deosebire de platformele AOP, care sunt folosite pentru dezvoltarea sistemelor descentralizate și al căror avantaj principal față de un limbaj de programare standard îl reprezintă facilitățile în ceea ce privește execuția multiplatformă, o platformă de tip ABMS

este un software de modelare în care agenții (reprezentând atât entitățile mobile, cât și mediul) interacționează pe baza unui set explicit de reguli. Este o alternativă la modelarea matematică a unui sistem sau la efectuarea unui experiment în lumea reală. Motivele pentru care ABMS câștigă popularitate în rândul disciplinelor care nu sunt legate de dezvoltarea software sunt simplitatea și flexibilitatea acestuia în ceea ce privește definirea unui sistem distribuit, oferind în același timp descrierea naturală a acestuia, captând și comportamentul său emergent (Odell, 2011). Pe baza unui studiu amănunțit al literaturii disponibile, platformele ABMS (Kalliopi et al., 2015) sunt mai numeroase și tind să fie mai actualizate, în principal pentru că sunt folosite de o gamă mai largă de cercetători. În continuare este prezentată o listă a celor mai citate platforme ABMS, cu câteva observații asociate fiecăreia dintre ele.

Tabelul 1.2 Comparație între diferite platforme de tip ABMS

Platformă ABMS	Ultima versiune	Limbaj de programare	Observații
Netlogo (Macron et al., 2017)	version 6, 2021	NetLogo, necesită Java	Mediu de programare și modelare SMA, gratuit și ușor de învățat cu care se pot studia fenomene de natură emergentă.
MaDKit (Gutknecht et al., 2000)	version 5, 2021	Java	Bibliotecă Java cu amprentă redusă pentru simulare și modelare.
Mesa (Simoiu et al., 2022)	2022, GitHub	Python	Cadru de dezvoltare a aplicațiilor multi-agent cu aplicații în modelarea proceselor, dezvoltat în Python cu următoarele funcționalități: componente modulare, vizualizare în browser, unelte de analiză încorporate.
Repast (North et al., 2013)	Version 2.9.1, 2022	Java, C++, Python	Aplicație de modelare de tip open-source care este în continuă dezvoltare. Poate fi folosit atât pe stația de lucru personală cât și în infrastructuri de tip cluster.
AnyLogic (Murarev et al., 2021)	Ver. 8.7.8, 2022	Limbaj de modelare grafic și Java	Platforma de modelare disponibilă atât în versiune gratuită cât și comercială. Această platformă este folosită pentru dezvoltarea de SMA și de simulări bazate pe evenimente cu aplicații în domenii conexe conceptului de lanț de aprovizionare.

4. Formalismul multi-agent în conducerea proceselor de producție

Această secțiune tratează integrarea agenților software inteligenți cu dispozitive de automatizare de nivel scăzut, un concept întâlnit în literatura de specialitate referitoare la SMA

ca agenți industriali. Un concept relativ nou (IEEE, 2020), agenții industriali se referă la legătura dintre nivelul informațional și cel fizic, legatură care furnizează dispozitivelor de control de nivel scăzut, cum ar fi automate programabile (*programmable logic controller* – PLC), funcționalități de nivel înalt, precum algoritmi de optimizare, soluții de inteligență artificială (AI), protocoale de interacțiune etc.

Evoluția către legătura fizico-informațională (Răileanu, 2011) și materializarea acesteia în agenți industriali a pornit de la extinderea influenței agenților software către resursele fizice, ceea ce a rezultat în termenul de nișă holon (Derigent et al., 2021; Valckenaers et al., 2015), întâlnit în mod tradițional în controlul producției (Cardin et al., 2015) și termenul mai generic sistem ciber-fizic (Leitão et al., 2016). În consecință, această secțiune va prezenta aplicații ale formalismului SMA în conducerea producției și extinderea acestuia la managementul lanțurilor de aprovizionare cu funcționalități precum monitorizarea și trasabilitatea produselor.

a. Optimizare: alocarea operațiilor pe resurse

Până la a 4-a Revoluție Industrială, sistemele de conducere și control al producției au fost influențate de structurile organizaționale, rezultând arhitecturi de conducere cu ierarhii stricte de tip top-down (comenzile sunt calculate la un nivel superior, apoi sunt transmise către nivelul inferior pentru implementare). Cu toate acestea, în economia curentă (caracterizată de cereri greu de anticipat și cu evoluție rapidă) este preferată o abordare mai dinamică pentru a face față schimbărilor frecvente într-un mod independent față de nivelurile superioare. În acest sens, formalismul SMA este adoptat prin asocierea agenților la entitățile implicate în procesul de conducere. Acești agenți îndeplinesc sarcini locale pe baza recomandărilor de nivel superior (Patch et al., 2012), una dintre ele fiind alocarea descentralizată a operațiunilor pe resurse. Un protocol standard de interacțiune în SMA, bazat pe regulile pieței (vânzători și cumpărători/participanți și inițiatori (*www.fipa.org*)), este utilizat pentru realizarea acestui proces de optimizare – *Contract Net Protocol* (CNP, (Smith, 1980)).

Proiectat inițial pentru alocarea sarcinilor pe resurse (Fig.1.9 sus), protocolul a fost adaptat pentru a lua în considerare unele aspecte particulare ale sistemelor de producție precum: a) diferențierea operațiunilor în procesare și transport, b) faptul că, pentru a executa un produs final sunt necesare mai multe operațiuni de procesare și între acesta pot apărea precedențe; procesul trebuie reiterat în anumite situații, c) răspunsurile de la resurse și timpul de execuție a unei operațiuni nu este instantaneu (Fig.1.9 jos).

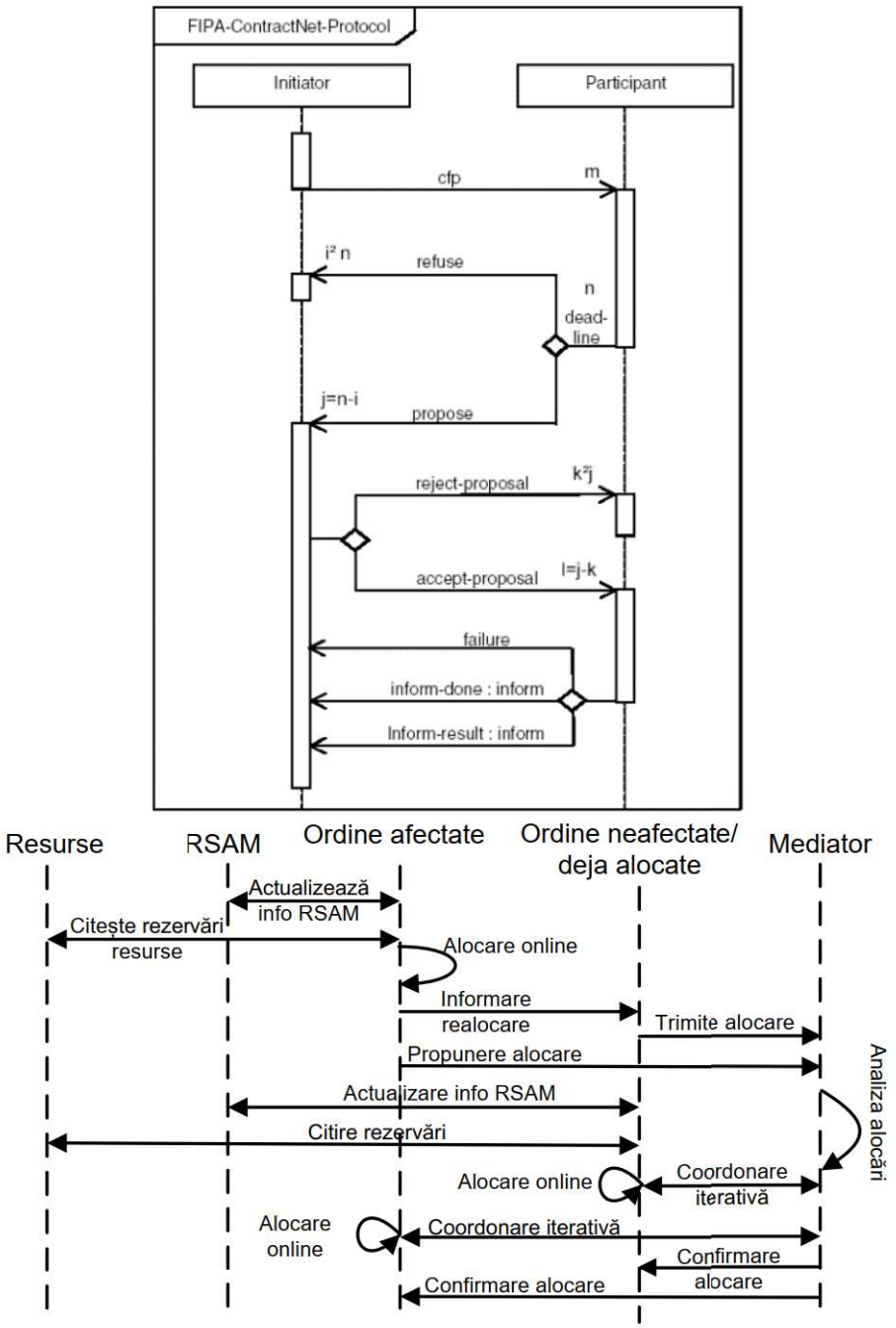

Fig.1.9 Comparație între protocolul CNP original (FIPA, 2022; Smith, 1980) și adaptarea sa pentru alocarea operațiilor pe resurse în cadrul unui sistem de fabricație (Borangiu et al., 2011)

b. Configurarea grupurilor de resurse pentru procesul de producție

Un alt mecanism utilizat împreună cu alocarea descentralizată a operațiilor este selectarea resurselor necesare pentru a executa un lot de produse. Această problemă a fost

rezolvată de arhitectura CoBASA (*Coalition Based Approach for Shop floor Agility*) (Barata et al., 2003). În acest sens, agenți software sunt asociați resurselor fizice extinzând operațiunile pe care în cele din urmă le pot executa fizic cu capacități software, precum negocierea, contractarea și service-ul, capabile să participe la coaliții/consorții. Rezultatul acestui proces este un agent de tip resursă (*Manufacturing Resource Agent* – MRA). Mai multe MRA-uri se pot asocia, formând o coaliție, pe baza cerințelor de producție și sub îndrumarea unui agent coordonator (*Coordinator Agent* – CA) pentru a genera funcționalități agregate dincolo de capacitățile individuale ale fiecărui agent component. Acest proces este descris în figura 1.10.

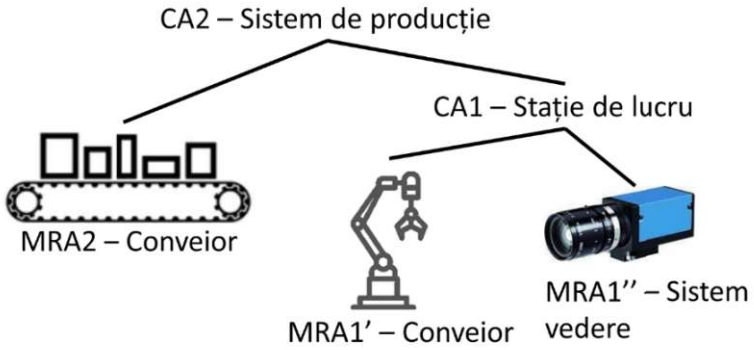

Fig.1.10 Formarea unei coaliții de agenți de fabricație (inspirat din (Barata et al., 2003))

c. Rutarea dinamică a produselor în curs de fabricație

O sarcină importantă a unui sistem de conducere a fabricației este rutarea produselor către posturile de lucru selecționate pentru a efectua operații de procesare. În prezent, datorită progreselor în electronică, este posibilă asocierea dispozitivelor încorporate cu capacități decizionale, direct entităților care compun fluxul de produse în curs de fabricație (*Work in Process* – WIP). Astfel, a apărut conceptul de produs activ (Sallez et al., 2009) sau produs inteligent (McFarlane et al., 2013; Wong et al., 2002). Acesta este compus din produsul pasiv cu un modul decizional de augmentare asociat, executând un agent responsabil de luarea deciziilor locale. Autorii din referința (Sallez et al., 2009) prezintă un sistem în care tehnologia SMA este utilizată pentru măsurarea și actualizarea timpilor de transport în timp real; aceste informații sunt utilizate pentru rutarea adaptivă: agenții mobili care reprezintă produsele măsoară și partajează informații pentru a alege dinamic o cale care minimizează timpul de transport. O descriere grafică a acestui proces împreună cu localizarea agenților (strâns cuplați și pe dispozitiv conform (IEEE, 2020)) este descrisă în figura 1.11.

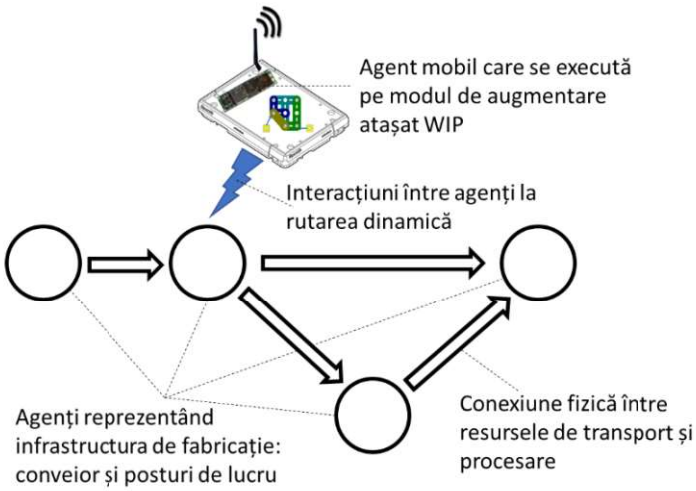

Fig.1.11 Soluție multi-agent pentru rutarea inteligentă a produselor în cadrul unui sistem de fabricație (Sallez et al., 2009)

d. Trasabilitatea produselor în curs de fabricație

O tendință actuală în conducerea fabricației este conceptul de fabricație la comandă (*Make-to-Order* – MTO) (Roehrich et al., 2011), care se caracterizează prin rolul clientului în generarea comenzilor și, de asemenea, posibilitatea acestuia de a le personaliza și urmări. În acest sens, tehnologiile SMA sunt folosite aici pentru a urmări comanda, atât în etapa de fabricație (Meyer et al., 2009), cât și pe parcursul întregului lanț de aprovizionare (Wong et al., 2002). Din punct de vedere practic, tehnologiile actuale precum RFID (autoidentificare), comunicațiile fără fir, senzorii și acționările miniaturizate permit virtualizarea evenimentelor fizice și, prin urmare, a celor ale produselor în fabricație și, de asemenea, a produselor finale, permițând totodată clientului să influențeze în mod dinamic modul în care comanda este produsă, stocată sau transportată. Conceptul cheie aici este „produs inteligent“, care reprezintă o actualizare a produsului fizic clasic. Această inteligență poate fi înțeleasă în două moduri: un produs cu informații inteligente (informațiile sunt virtualizate și procesate în puncte fixe de agenți reprezentând agenți de producție sau de aprovizionare, cum ar fi resursele de procesare sau manipulare) sau un produs cu inteligență îmbarcată. În al doilea caz, inteligența este reprezentată de un agent situat local sau la distanță (Meyer et al., 2009; Mcfarlane et al., 2013; Wong et al., 2002). Avantajul unui agent asociat este că poate produce evenimente de interes pentru ciclul de viață al produsului asociat, atât pe baza locației sale fizice (determinată, de exemplu, dintr-un sistem de autoidentificare, folosind RFID sau un sistem de identificare echivalent), cât și a istoricului și a stării sale interne stocate într-o bază de date. O descriere grafică a unui astfel de sistem este dată în figura 1.12.

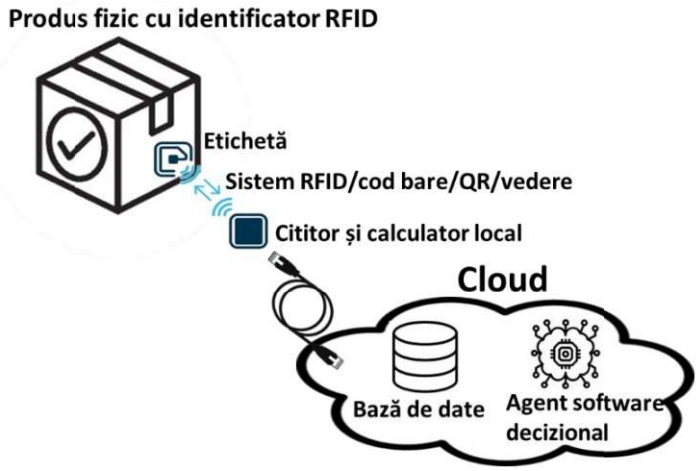

Fig.1.12 Produs inteligent cu capacități decizionale „la distanță“ implementat folosind tehnologia SMA(adaptat din (Wong et al., 2002)

5. Concluzii și perspective

După cum a fost menționat în introducere, SMA (v. Fig.1.3) câștigă din ce în ce mai mult teren în proiectarea, analiza, simularea și dezvoltarea proiectelor descentralizate. Cu toate acestea, în secțiunea următoare, o comparație între diferitele ramuri ale SMA, și anume AOP (Tabelul 1.1) și ABMS (Tabelul 1.2), subliniază că proiectele care implică simulare și, în consecință, instrumentele ABMS sunt mai numeroase, mai actualizate și mai utilizate. Pe scurt, în ceea ce privește agenții industriali, există platforma JADE cu extensiile sale (WADE și JADEX) și câteva proiecte limitate care implică JACK/Erlang. În ceea ce privește seturile de instrumente de modelare și simulare SMA, acestea sunt în general aplicații de sine stătătoare (de exemplu: NetLogo, Swarm, ș.a., v. Tabelul 1.2). În acest sens, puterea SMA constă în capacitatea sa de a modela un sistem descentralizat, atât din punct de vedere structural (agenți cu roluri și locații asociate), cât și din punct de vedere dinamic (protocoale de interacțiune și standardizarea mesajelor schimbate, v. FIPA), făcându-l mai degrabă un formalism, și nu un instrument. Acest lucru este subliniat de faptul că proiectele descentralizate pot fi implementate fără a menționa termenul agent. Aceste proiecte conțin doar aplicații software care rulează pe resurse separate, aplicații dezvoltate în limbaje de programare dedicate, singura cerință fiind ca aceste aplicații să se înțeleagă între ele (standardizarea protocoalelor de comunicare, ceea ce face posibilă comunicarea între PLC-uri și dispozitivele încorporate). Un alt avantaj al platformelor SMA AOP este că oferă posibilitatea implementării cu ușurință a strategiilor de conducere bazate pe stări, prin utilizarea conceptului de comportament (de exemplu, comportamente secvențiale, paralele, FSM – JADE). Observația anterioară este întărită de

faptul că toate proiectele prezentate aici folosesc platformele AOP ca strat decizional care este completat/extins de aplicații dedicate ce fac legătura între stratul informațional și cel fizic. În acest sens, agentificarea unui robot industrial presupune asocierea unui PC/sistem embedded dedicat care execută agentul industrial datorită faptului practic că dezvoltatorii nu au acces la sistemul de operare al controlerului sau în unele cazuri – precum PLC-urile – nici măcar nu este posibil să se ruleze cod extern precum cel aferent platformei AOP pe acel sistem. Autorii lucrărilor citate în această secțiune, cu aplicații în producție și energie electrică, afirmă importanța asocierii agenților software cu protocoale industriale pentru a reduce decalajul dintre nivelul informațional și cel fizic.

Unele concluzii parțiale includ: a) abordarea SMA va câștiga mai mult teren în aplicațiile industriale, în principal datorită miniaturizării dispozitivelor încorporate, permițând astfel distribuirea inteligenței și luarea locală a deciziilor, b) prin executarea agenților software direct pe dispozitivele de conducere și control, agenții pot opera în timp real, c) formalismul SMA este o soluție pentru implementarea sistemelor ciber-fizice și d) va fi posibilă măsurarea performanței agenților.

Capitolul 2: Prezentare JADE

Cuprins

1. Introducere

JADE (*Java Agent Development Framework*) (*https://jade.tilab.com/*) este un cadru de dezvoltare a aplicațiilor multi-agent (*framework*) implementat în limbajul Java. JADE facilitează dezvoltarea sistemelor multi-agent printr-un middleware conform cu specificațiile FIPA (www.fipa.org) împreună cu un set de unelte folosite la debugging și implementare. Sistemul multi-agent poate fi distribuit pe mașini (care pot rula diferite tipuri de sisteme de operare) și configurarea globală se poate face printr-o interfață vizuală în mod distant (*remote*). Configurarea poate fi alterată în timpul rulării prin relocarea agenților de pe o mașină pe alta în funcție de cum și când este nevoie. JADE este complet implementat în limbajul Java și

cerința minimală de rulare este versiunea 1.4 (pentru JRE – *Java Runtime Environment* – sau pentru JDK – *Java Development Kit*). Platforma JADE este dezvoltată de Telecom Italia și ultima versiune (la momentul redactării acestui material) este 4.6 (lansată în 2022).

Programarea orientată pe agenți (POA sau AOP – *Agent Oriented Programming*) este o paradigmă software relativ recentă, introdusă în anii '90 de către Yoav Shoham (Shoham, 1993), ce introduce concepte din teoriile inteligenței artificiale în domeniul sistemelor distribuite.

POA modelează o aplicație ca pe o colecție de componente denumite agenți, caracterizate de autonomie, pro-activitate și abilitate de comunicare. Fiind autonomi, agenții pot realiza sarcini complexe și de obicei de lungă durată în mod independent. Fiind pro-activi, aceștia pot lua inițiativa în realizarea unei anumite sarcini, fără a primi comenzi de la utilizator. Prin comunicare, agenții pot interacționa cu alte entități pentru a ajuta sau a fi ajutați în realizarea de sarcini. Modelul arhitectural al unei aplicații orientate pe agenți este de tip punct-la-punct, astfel încât fiecare agent poate iniția o comunicație cu orice alt agent sau poate fi subiectul unui apel de comunicație în orice moment.

Limbajele de programare orientate pe agenți reprezintă o nouă clasă de limbaje de programare ce pun accent pe caracteristicile principale ale sistemelor multi-agent. Un astfel de limbaj de programare trebuie să includă cel puțin o structură corespunzătoare agenților, dar multe limbaje oferă și mecanisme de sprijin pentru atribute adiționale ale agenților, cum sunt convingerile, scopurile, planurile, rolurile și normele lor (*Belief-Desire-Intent (BDI) agents*). În prezent avem la dispoziție multe astfel de limbaje de programare. Unele dintre ele sunt dezvoltate prin codificare directă a unor teorii ale agenților, în vreme ce altele extind limbaje existente pentru a le putea face compatibile cu necesitățile agenților.

Un rol important în dezvoltarea sistemelor multi-agent îl au platformele software și frameworkurile. Majoritatea permit implementarea sistemelor multi-agent pe diferite tipuri de hardware sau sisteme de operare, punând la dispoziție un middleware pentru a permite execuția lor și realizarea operațiilor esențiale agenților (comunicarea și coordonarea). Majoritatea acestor platforme și framework-uri oferă funcționalități compatibile cu FIPA (*Foundation For Intelligent, Physical Agents*) permițând astfel, interoperabilitate între mai multe sisteme multi-agent. Unele dintre ele permit lucrul pe diferite tipuri de hardware, de comunicare în rețea, de arhitecturi ale agenților (ex. JADE), în vreme ce altele sunt destinate unor tipuri speciale de agenți (ex. agenți mobili).

Atunci când se utilizează o abordare orientată pe agenți în automatizarea unor procese apar o mulțime de probleme dependente de domeniul de lucru, care trebuie rezolvate (ex.: modul de realizare a comunicației între agenți). Este mai ușor să se realizeze sisteme multi-agent folosind un middleware orientat pe agenți ce pune la dispoziție o infrastructură independentă de domeniul de lucru, permițând dezvoltatorilor să pună accent pe producție, nu pe implementarea elementelor de nucleu ale infrastructurii.

Unul dintre cele mai răspândite middleware-uri orientate pe agenți este JADE (*Java Agent Development framework*), un sistem complet distribuit cu o infrastructură flexibilă, ce permite adăugarea facilă de module adiționale. Cadrul de lucru JADE permite dezvoltatorilor realizarea unor aplicații bazate total pe agenți, prin intermediul unui mediu de execuție ce implementează caracteristicile necesare ciclului de viață al agenților, logica de bază a agenților și pune la dispoziție o gamă largă de instrumente grafice.

JADE este scris complet în Java și din acest motiv beneficiază de un set mare de funcții și de biblioteci, permițând dezvoltatorilor cu o expertiență minimă în domeniu să realizeze sisteme multi-agent în JADE. Inițial, JADE a fost dezvoltat de departamentul de cercetare și dezvoltare al Telecom Italia, dar este în prezent un proiect comunitar distribuit gratuit sub licența LGPL.

2. Platforma JADE și paradigma agenților

JADE este o platformă software care oferă funcționalități middleware, independente de aplicația specifică și care simplifică realizarea aplicațiilor distribuite ce exploatează noțiunea de agent software (Leitao et al., 2017). JADE implementează această abstractizare folosind limbajul orientat obiect Java, oferind o interfață software de programare (*application programming interface* – API) simplă și ușor de folosit. Următoarele alegeri de proiectare au fost influențate de abstractizarea la nivelul conceptului de agent:

- autonomie și proactivitate: Un agent nu poate furniza apeluri de tip call-back sau propria referință de obiect altor agenți din rațiuni de securitate – minimizarea posibilității ca alte entități să preia controlul serviciilor sale. Un agent trebuie să aibă propriul fir de execuție, folosindu-l pentru a controla ciclul său de viață și să decidă în mod autonom când să efectueze anumite acțiuni;
- cuplare slabă și posibilitatea de a „spune Nu“: Forma de bază de comunicare între agenți în JADE este o comunicare asincronă, bazată pe mesaje; un agent care dorește să

comunice trebuie să trimită un mesaj către o destinație identificată (sau un set de destinații). Prin destinație se înțelege un agent, parte a platformei multi-agent. Nu există o dependență temporală între expeditor și destinatari; este posibil ca un receptor să nu fie disponibil atunci când expeditorul trimite mesajul. De asemenea, nu este nevoie să se obțină referința obiectului agenților receptor, ci doar să se numească identitățile pe care sistemul de transport al mesajelor le poate rezolva în adrese de transport corespunzătoare (*agent identifier*). Este chiar posibil ca o identitate precisă a destinatarului să fie necunoscută expeditorului, care poate defini în schimb un set de destinatari ce utilizează o grupare intenționată (de exemplu, toți agenții care furnizează serviciul „Vânzare de cărți“) sau o mediere de un agent proxy (de exemplu, să propage acest mesaj tuturor agenților din domeniul “selling.book.it”);

- sistem descentralizat de tip „punct-la-punct“ (*Peer-to-Peer*): Fiecare agent este identificat printr-un nume unic la nivel de platformă (AgentIDentifier, sau AID, conform definiției FIPA). Acesta se poate alătura sau părăsi o platformă gazdă în orice moment și poate descoperi alți agenți atât prin intermediul serviciilor de tip Pagini albe (*White paper*, implementat de agentul AMS), cât și prin servicii de tip Pagini aurii (furnizate de agenții de tip DF). Un agent poate iniția comunicarea cu orice alt agent în orice moment și poate fi la fel destinatarul unui proces de comunicare în orice moment.

Platforma JADE a fost implementată pentru a oferi programatorilor următoarele funcționalități de bază, gata de utilizare și ușor de personalizat:

- un sistem complet distribuit compus din agenți autonomi. Fiecare agent rulează ca un fir de execuție separat, posibil pe diferite mașini separate, și este capabil să comunice transparent cu alți agenți. Platforma oferă un API unic, independent de locație, care abstractizează infrastructura de comunicație;
- respectarea deplină a specificațiilor FIPA. Platforma a participat cu succes la toate testele de interoperabilitate FIPA și a fost folosit ca middleware pentru multe platforme SMA. Un mare avantaj al acestui lucru a fost contribuția activă a echipei JADE la procesul de standardizare FIPA;
- implementări ale funcționalităților de tip Pagini albe (căutare agenți după nume) și Pagini aurii (căutare agenți după serviciile oferite);
- transport eficient al mesajelor asincrone printr-un API transparent ca localizare. Platforma selectează cele mai bune mijloace de comunicare disponibile și, atunci când este posibil, evită împachetarea/dezpachetarea (*marshalling*) obiectelor Java. La

trecerea limitelor platformei, mesajele sunt transformate automat din propria reprezentare Java internă a JADE în sintaxa standard FIPA cu tot cu modalitatea de codificare și protocoalele de transport asociate;

- un simplu, dar eficient, mod de gestionare a ciclului de viață al unui agent. Când se creează agenți, li se atribuie automat un identificator unic global și o adresă de transport care sunt utilizate pentru a se înregistra la serviciul de pagini albe al platformei. Platforma deține un API și instrumente grafice care operează atât la nivel local cât și la distanță (*remote*) pentru a gestiona ciclul de viață al agenților, de exemplu, creare, suspendare, reluare, înghețare (*freeze*), migrare/relocare, clonare și distrugere;
- suport pentru mobilitatea agenților. Atât codul agentului, cât și, sub anumite restricții, starea agentului, pot migra între procese și mașini. Migrarea agenților este transparentă pentru ceilalți agenți ai platformei, care pot continua să interacționeze chiar și în timpul procesului de migrare;
- un mecanism de abonare pentru agenți și chiar aplicații externe care doresc să se aboneze la o platformă pentru a fi notificați cu privire la toate evenimentele platformei, inclusiv evenimentele legate de ciclul de viață și cele care implică schimbul de mesaje;
- un set de instrumente grafice pentru depanare și monitorizare. Acestea sunt deosebit de importante și complexe, fiind de un real ajutor în proiecte multi-threaded, multi-proces, multi-mașină, așa cum este cazul proiectelor JADE standard. Conversațiile între agenți pot fi monitorizate, iar execuția agentului poate fi controlată de la distanță, inclusiv depanarea pas cu pas a execuției agentului;
- suport pentru ontologii și limbaje de codificare a conținutului (*content languages*). Verificarea ontologiei și codificarea conținutului se efectuează automat de către platformă, programatorii putând selecta limbajul de codificare conținut și ontologiile preferate (de exemplu, XML și RDF);
- o bibliotecă de protocoale de interacțiune care modelează șabloane tipice de interacțiune orientate către atingerea unuia sau mai multor obiective. Șabloane generice, independente de aplicație, sunt disponibile ca un set de clase Java care pot fi personalizate cu cod specific soluției. Protocoalele de interacțiune pot fi, de asemenea, reprezentate și implementate ca un set de mașini de stare concurente;
- integrarea cu diverse tehnologii Web, inclusiv JSP, servlets, applets și tehnologii de tip servicii Web. Platforma poate fi, de asemenea, configurată pentru a trece de firewall-uri și de a folosi sisteme NAT;

- suport pentru platforma J2ME și comunicație wireless. Mediul de rulare (*run-time*) JADE este disponibil pentru platformele J2ME-CDC și -CLDC printr-un set uniform de API-uri care acoperă atât mediile J2ME, cât și J2SE;
- o interfață pentru lansarea/controlul unei platforme și a componentelor sale distribuite dintr-o aplicație externă;
- un nucleu extensibil proiectat pentru a permite programatorilor să extindă funcționalitatea platformei prin adăugarea de servicii distribuite la nivel de nucleu. Acest mecanism este inspirat de abordarea de programare orientată spre aspect, în care diferite aspecte pot fi incluse în codul de aplicație și coordonate la nivel de nucleu. Pentru a menține compatibilitatea cu mediul J2ME care nu acceptă în mod intrinsec codul orientat spre aspect, JADE utilizează o tehnică proprie de filtrare.

3. Arhitectura JADE

Figura 2.1 prezintă principalele elemente arhitecturale ale unei platforme JADE. O platformă JADE este compusă din containere agent care pot fi distribuite în rețea. Agenții sunt grupați în containere, care sunt procese Java, ce oferă run-time-ul JADE și toate serviciile necesare pentru găzduirea și executarea agenților.

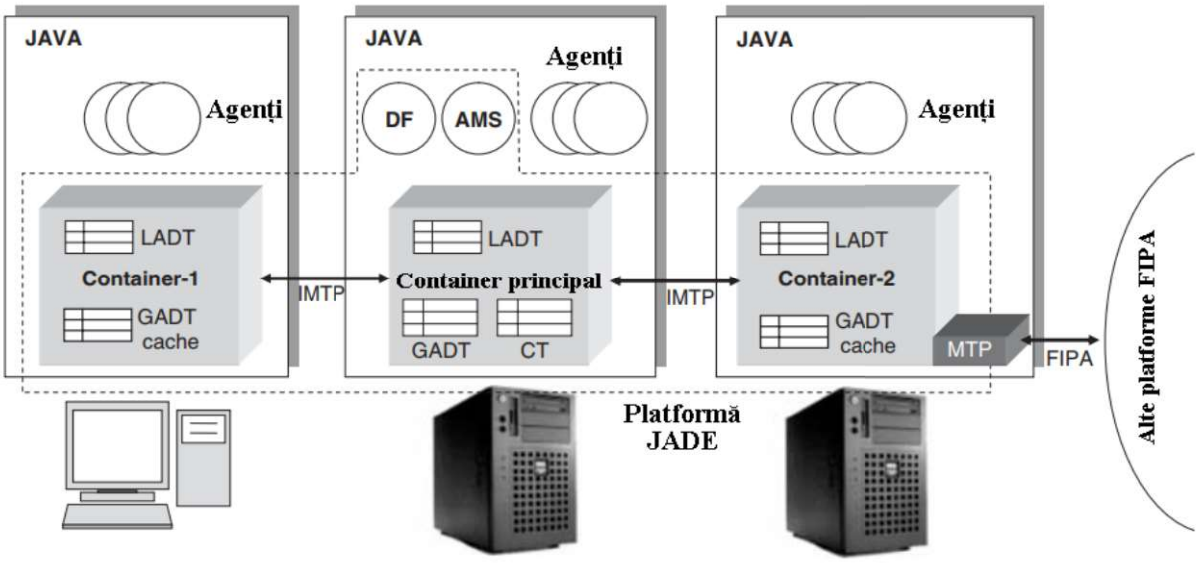

Fig.2.1. – Arhitectura unei platforme JADE

Există un container special, numit container principal (*Main-Container*), care reprezintă punctul de start al unei platforme; este primul container care urmează să fie lansat și toate celelalte containere trebuie să se atașeze la un container principal prin înregistrare. Diagrama UML din figura 2.2 arată relațiile dintre principalele elemente arhitecturale ale JADE.

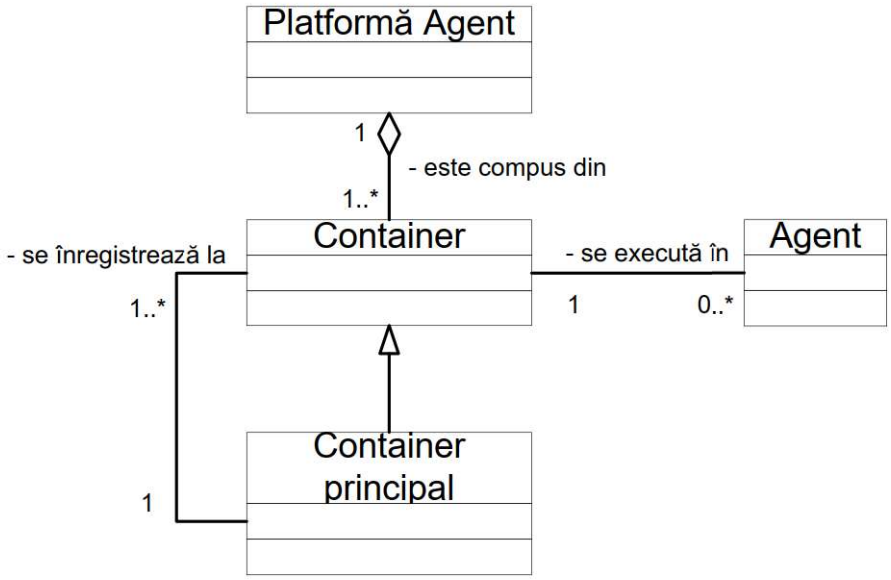

Fig.2.2 – Relațiile dintre principalele elemente arhitecturale JADE

Identificarea containerelor se face utillizând un nume logic; în mod implicit, containerul principal este denumit *main container*, în timp ce celelalte sunt numite "Container-1", "Container-2" etc. Numele implicite pot fi modificate folosind opțiunile din linia de comandă. Ca punct de start, containerul principal are următoarele funcții:

- gestionarea tabelului de containere, care este registrul referințelor obiectelor SMA și al adreselor de transport ale tuturor nodurilor container care compun platforma;
- gestionarea tabelului global ce conține descriptorii de agenți (GADT), care este registrul tuturor agenților prezenți în platformă, inclusiv starea și locația lor actuală;
- găzduirea AMS și DF responsabili cu gestionarea agenților și a serviciilor oferite de platformă.

În mod implicit containerul principal al platformei JADE este un punct unic de defect. În practică, situația nu stă astfel pentru că platforma JADE realizează o replicare a GADT pe fiecare container, replica ce este gestionată la nivel local. Operațiunile platformei, în general, nu implică containerul principal, ci doar memoria cache locală și cele două containere care găzduiesc agenții expeditor și destinatar ai mesajului. Atunci când un container trebuie să descopere unde este găzduit destinatarul unui mesaj, acesta caută mai întâi în LADT (tabelul descriptor al agentului local) și apoi, numai în cazul în care căutarea nu reușește, este contactat containerul principal pentru a obține referința la agentul dorit care, în consecință, este memorată în cache-ul local pentru utilizări viitoare. Deoarece sistemul este dinamic (agenții

pot migra, pot fi terminați sau se închid în mod programat, ori pot fi creați agenți noi), ocazional sistemul poate utiliza o valoare învechită, rezultând o adresă incorectă (nonvalidă). În acest caz, containerul primește o excepție și este forțat să reactualizeze memoria cache a containerului principal. Politica de înlocuire a memoriei cache este de tip LRU (datele cel mai puțin utilizate recent), care a fost proiectat pentru a optimiza conversații lungi, mai degrabă decât sporadice, caracterizate prin schimburi de mesaje unice, acestea fiind de fapt destul de puțin frecvente în proiecte multi-agent.

Cu toate acestea, chiar dacă containerul principal nu este un punct de blocaj, acesta rămâne un punct unic de defect. Pentru a gestiona acest lucru se poate utiliza Serviciul principal de replicare (*Main Replication Service* – MRS) pentru a se asigura că platforma JADE rămâne pe deplin operațională chiar și în cazul unei defectări a containerului principal. Cu acest serviciu, administratorul poate implementa toleranța la defecte a platformei, nivelul de scalabilitate și nivelul de distribuție al platformei. Un nivel de control intermediar compus din mai multe instanțe distribuite ale containerului principal poate fi configurat pentru a implementa un sistem de pornire distribuit (adică mai multe puncte de pornire pot fi transmise fiecărui container). În caz extrem, fiecare container poate fi configurat să se alăture Serviciului principal de replicare – MRS și să acționeze ca parte a nivelului de control.

Identitatea agentului este conținută într-un identificator de agent (AID – *Agent IDentifier*), alcătuit dintr-un set de componente care respectă structura și semantica definite de FIPA. Elementele de bază ale AID sunt numele agentului și adresele acestuia. Numele unui agent este un identificator unic global pe care JADE îl construiește prin concatenarea numelui instanței dat de utilizator (cunoscută și ca nume local, suficientă pentru a nu exista ambiguități în comunicarea intraplatformă) cu numele platformei. Adresele agentului sunt adresele de transport moștenite de platformă, unde fiecare adresă de platformă corespunde unui punct final MTP (*Message Transport Protocol*) unde pot fi trimise și primite mesaje compatibile FIPA. Dezvoltatorii au posibilitatea de a-și adăuga propriile adrese de transport la AID atunci când doresc să implementeze propriul protocol de comunicație agent.

La lansarea containerului principal, se instanțiază automat doi agenți speciali care sunt lansați de JADE, rolurile acestor doi agenți fiind definite de standardul de gestiune al agenților FIPA:

a. Sistemul de Management al Agenților (*Agent Management System* – AMS) este agentul care supraveghează întreaga platformă; este punctul de contact pentru toți agenții care

trebuie să interacționeze pentru a accesa paginile albe ale platformei, precum și pentru a gestiona ciclul lor de viață. Fiecare agent trebuie să se înregistreze la AMS (se realizează automat de JADE la pornirea agentului) pentru a obține un AID valid;

b. Sistemul de tip pagini aurii (*Directory Facilitator* – DF) este utilizat de orice agent care dorește să își înregistreze serviciile sau să caute alte servicii disponibile. JADE DF acceptă, de asemenea, înregistrări de la agenți care doresc să fie notificați ori de câte ori se face o înregistrare de serviciu sau o modificare ce corespunde unor criterii specificate. Pentru a distribui serviciul de Pagini aurii în mai multe domenii pot fi pornite simultan mai multe DF-uri. Acestea pot fi sincronizate prin stabilirea de înregistrări încrucișate între ele, care permit propagarea cererilor efectuate de agenți.

4. Lansarea platformei din linia de comandă

Software-ul legat de JADE este împărțit în două secțiuni: distribuția principală și programe conexe. Acestea pot fi descărcate de la adresa *http://jade.tilab.com*. Distribuția principală este compusă din patru arhive cu următorul conținut:

- jadeBin.zip conține fișierele gata compilate JADE (Java *.jar), gata de utilizare;
- jadeDoc.zip conține documentația, inclusiv ghidurile Administrator și Programator;
- jadeExemples.zip conține codul sursă pentru diferite exemple;
- jadeSrc.zip conține toate sursele JADE.

Structura de subdirectoare arată ca în figura 2.3 și conține următoarele elementele principale:

- fișierul `jade/doc/index.html` care conține linkuri către o varietate de ghiduri tematice, ghidul programatorului și administratorului, documentația javadoc a tuturor surselor, plus alte câteva documente suport;
- subirectorul `jade/lib` conține toate fișierele *.jar care trebuie incluse în Java CLASSPATH pentru a rula JADE. Acesta include subdirectorul `lib/commons-codec` unde este distribuit un codec Base64 extern care trebuie să fie, de asemenea, inclus în Java CLASSPATH. În versiunea actuală (4.6) a mai rămas doar `jade.jar`.

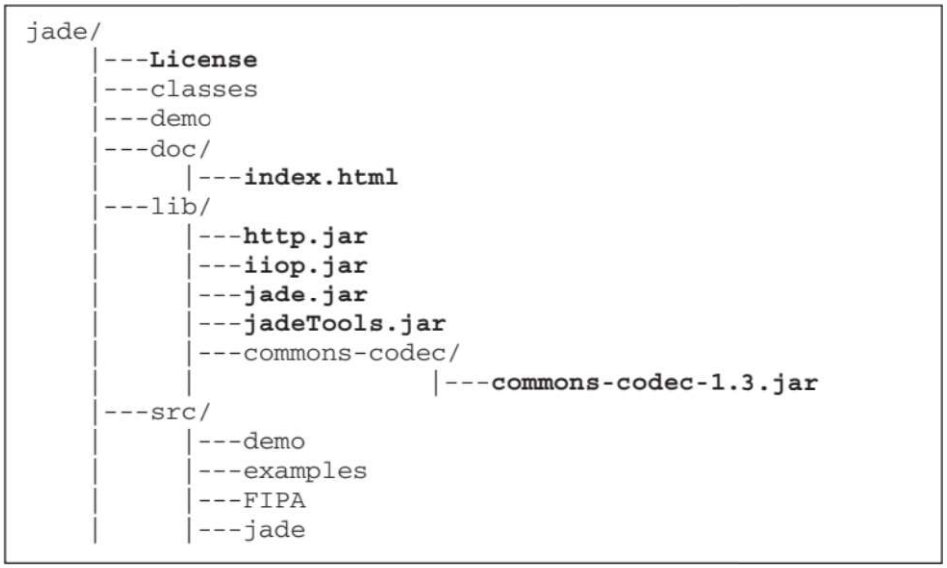

Fig.2.3 – Structura de subdirectoare JADE

- directorul `jade/src` conține patru subdirectoare: demo, examples, FIPA, jade în care se află coduri sursă pentru diferite tipuri de agenți, un modul definit de FIPA și toate sursele JADE. În versiunea actuală (4.6) a mai rămas doar `jade.jar`, codec-urile găsindu-se acum în calea `\jade\lib\commons-codec`.

Sursele JADE pot fi compilate cu ajutorul instrumentului ANT. Cele mai importante targeturi ANT sunt următoarele:

- jad – pentru a compila sursele și pentru a crea fișiere de clasă în cadrul subdirectorului de clase;
- lib – pentru a genera fișiere de arhivă Java în subdirectorul lib;
- doc – pentru a genera fișierele de documentație javadoc în subdirectorul doc;
- examples – pentru a compila toate exemplele.

Pentru a lansa platforma, trebuie setată mai întâi variabila locală de mediu (Java) CLASSPATH, adică setul de directoare și fișiere de arhivă Java unde Mașina Virtuală Java va căuta fișiere *.class și *.jar. Pentru a încărca un agent pe platformă, fișierul său obiect trebuie să fie accesibil prin intermediul CLASSPATH; de aceea se recomandă adăugarea căii curente la variabila de mediu. Dacă, de exemplu, JADE a fost descărcat în C:\JADE pe o platformă Windows, CLASSPATH poate fi setat utilizând următoarea secvență în linia de comandă:

```
prompt> set JADE_HOME=c:\jade
```

```
prompt> set CLASSPATH=%JADE_HOME%\lib\jade.jar; %JADE_HOME%
\lib\jadeTools.jar; %JADE_HOME%\lib\http.jar; %JADE_HOME% \lib\iiop.jar;
%JADE_HOME% \lib\commons-codec\commons-codec-1.3.jar;%JADE_HOME%\classes
```

Elementele necesare pentru rularea platformei JADE sunt: Java Development Kit (JDK) pentru realizarea aplicațiilor și compilarea lor, și biblioteca JADE. Opțional, este recomandată utilizarea unui mediu de dezvoltare integrat (ex.: Eclipse – *https://www.eclipse.org/*) care facilitează operarea cu căile de compilare și execuție (variabilele de mediu, CLASSPATH pentru biblioteci și fișiere compilate, și PATH pentru executabile). Din interfața vizuală, pentru un sistem care rulează Windows 10, pot fi urmați pașii din figura 2.4 pentru actualizarea variabilelor de mediu: click dreapta pe butonul de Start –> Settings –> System –> About –> Advanced system settings –> Advances –> Environment variables –> variabile mediu pentru utilizator, respectiv pentru sistem.

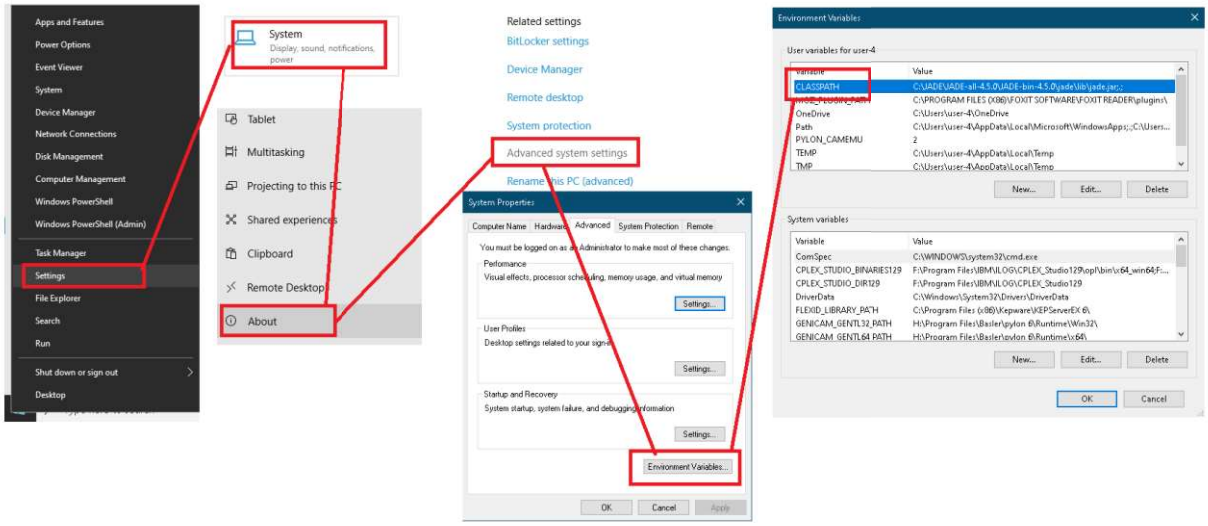

Fig.2.4 – Actualizarea variabilei de mediu CLASSPAH

Verificarea corectitudinii inserției câmpurilor dorite în variabilele de mediu se poate face din linia de comandă după resetarea consolei (închis/deschis) și apoi lansarea platformei. Se recomandă adăugarea în variabila de mediu Path a aplicației de compilare *javac*, precum și a căii curente - . (punct).

Containerul principal poate fi lansat acum cu GUI JADE folosind comanda:

```
prompt>javajade. Boot-gui
```

Rezultatul trebuie să fie așa cum se arată în figura 2.5.

```
prompt> java jade.Boot -gui
29-dic-2005 16.41.11 jade.core.Runtime beginContainer
INFO: -------------------------------------
    This is JADE snapshot - revision 5752 of 2005/07/15 14:22:11
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
-------------------------------------
29-dic-2005 16.41.14 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
29-dic-2005 16.41.15 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
29-dic-2005 16.41.15 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
29-dic-2005 16.41.15 jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
29-dic-2005 16.41.16 jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://anduril:7778/acc
29-dic-2005 16.41.19 jade.core.AgentContainerImpl joinPlatform
INFO: -------------------------------------
Agent container Main-Container@JADE-IMTP://NBNT2004130496 is ready.
```

Responsabilitate JADE

Inițializare servicii

Adrese MTP

Nume container

Fig.2.5 – Captura ecran după lansarea JADE.

Prima parte – responsabiliăți (*JADE disclaimer*) – apare la pornirea mediului de lucru JADE (*run-time*). După aceea, sunt inițializate toate serviciile standard ale platformei JADE, care implementează diferitele funcționalități furnizate de container. Deoarece instanța JADE este un container principal, se pornește în mod implicit un server HTTP – JADE și se afișează adresa sa locală. În final, o notificare indică faptul că un container numit „container principal“ este gata; platforma JADE este acum gata de utilizare. În mod implicit, platforma pornește pe portul 1099; adăugarea unei platforme pe un port deja utilizat nu este posibilă.

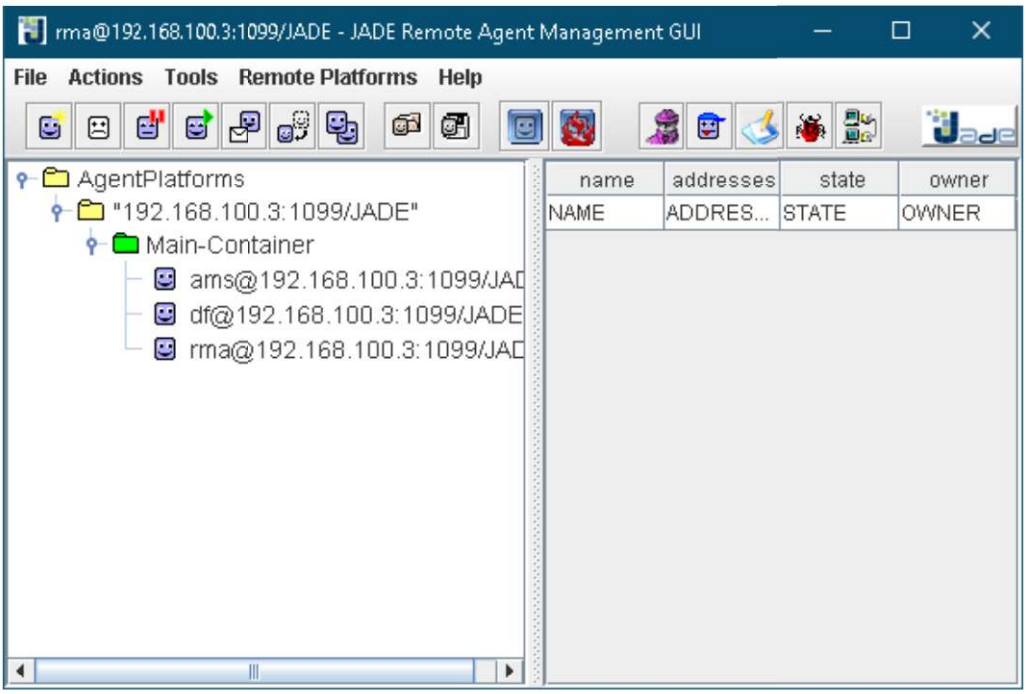

Fig.2.6 – Interfață GUI pentru JADE RMA

Opțiunea -**gui** din linia de comandă are ca efect lansarea interfeței grafice JADE, prezentată în figura 2.6. Această interfaţă grafică este un agent de sistem JADE numit Agent de monitorizare la distanță (*Remote Monitoring Agent* – RMA) și permite unui administrator de platformă să manipuleze și să monitorizeze platforma care rulează.

```
prompt> java jade.Boot -container -gazdă proces
```

Utilizarea RMA GUI, precum și toate celelalte instrumente grafice, poate avea un impact negativ asupra performanțelor sistemului. Acesta este unul dintre motivele pentru care este furnizată opțiunea -**gui**. Dacă performanța este critică, se recomandă ca RMA GUI să nu fie utilizată la implementare și să se limiteze utilizarea acesteia doar la monitorizarea sistemului.

După inițializarea containerului principal, se pot lansa alte containere pe diferite gazde care compun platforma. Dacă, de exemplu, numele mașinii gazdă în care a fost lansat containerul principal este „proces“, următoarea comandă va lansa un nou container pe gazda curentă și îl va atașa la containerul principal care rulează pe gazda specificată de -host.

```
prompt> java jade.Boot -container -host process
```

5. Lansarea platformei din Eclipse

Eclipse este un mediu de dezvoltare open-source realizat pentru a dezvolta preponderent aplicații în Java. Se recomandă descărcarea aplicației de la adresa https://www.eclipse.org/downloads/, unde se va opta pentru descărcarea pachetelor individuale (*Download Packages*) și apoi opțiunea Eclipse IDE for Java Developers, cu versiunea dorită: Windows, Linux sau macOS. Aceasta poate fi folosită pentru a dezvolta aplicații Java și, prin intermediul unor plug-in-uri, în alte limbaje, cum ar fi C, C++, COBOL, Python, Perl și PHP. Conceptul de plug-in este esențial în Eclipse, cu această opțiune fiind posibilă extensia funcționalităților de bază (ex.: adăugarea unui modul de realizare de interfețe grafice în Java – *WindowBuilder*). Proiectul Eclipse este actualizat în mod constant (pe parcursul ultimilor ani au fost lansate patru actualizări anuale) și pentru o bună funcționare împreună cu kitul de dezvoltare Java (*Java Development Kit* – JDK) este recomandat ca ambele produse să fie folosite cu versiunile la zi.

Platforma JADE poate fi lansată din mediul de dezvoltare integrat (IDE) Eclipse ca o aplicație Java. Pentru a putea realiza acest lucru, trebuie adăugat la proiectul aferent biblioteca

`jade.jar` (Fig.2.7). Ulterior, comanda de execuție se realizează sub forma unei configurații de rulare (Run -> Run configurations -> Java Application -> New) la care trebuie configurat proiectul (aici se selectează proiectul care implementează soluția multi-agent) și clasa principală (jade.Boot) (Fig.2.8). Dacă se dorește previzualizarea liniei de comandă, se selectează opțiunea "Show Command Line" (Fig.2.8, dreapta jos). Trebuie avut grijă la selectarea clasei principale pentru că în urma specificării acesteia proiectul pentru care se face configurația de rulare se poate schimba în mod automat și fără notificare. În secțiunea Arguments se configurează opțiunile de lansare și agenții rulați (Fig.2.9).

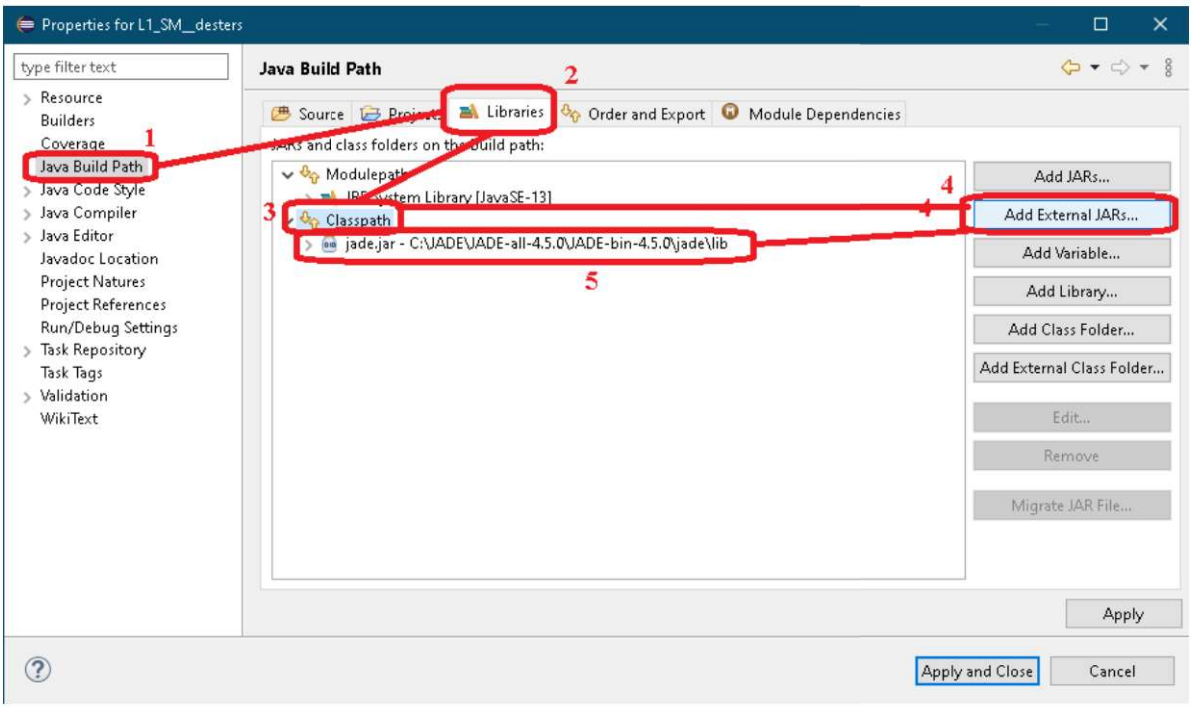

Fig.2.7 – Adăugarea bibliotecii JADE la un proiect Java

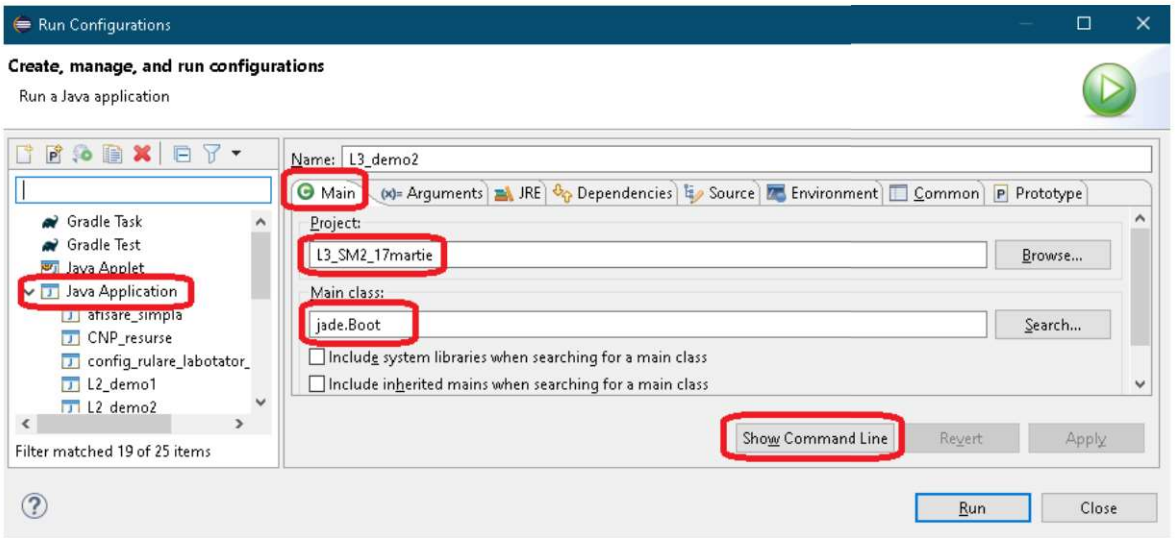

Fig.2.8 – Crearea unei configurații de rulare în Eclipse

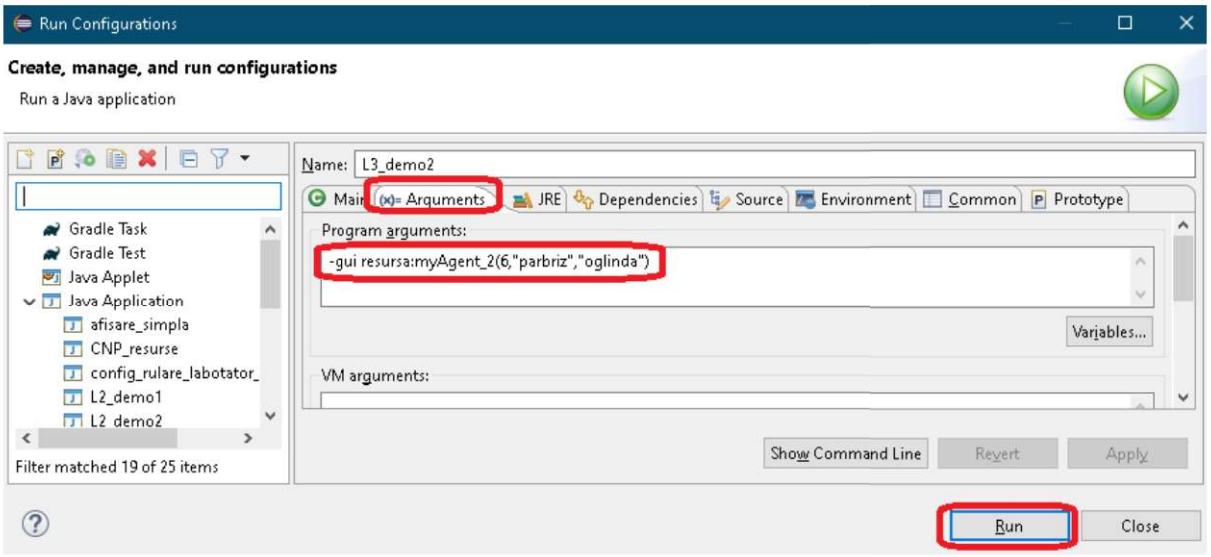

Fig.2.9 – Configurarea argumentelor de rulare și a agenților din Eclipse

6. Probleme des întâlnite

În urma rulării unei linii de comandă care lansează platforma JADE împreună cu agenții asociați pot apărea diferite probleme. În continuare sunt enumerate cele mai des întâlnite probleme și sunt explicate modalitățile de detecție și rezolvare:

- **Crearea de platforme multiple**. Platforma JADE este reprezentată de un server care operează în mod implicit pe portul 1099. Pornirea secvențială a mai multor platforme pe același port va rezulta în imposibilitatea rulării ultimelor/ultimei platforme. Acest lucru este vizibil în linia de comandă (terminal sau terminal Eclipse) prin mesajul: `Error adding ICP jade.imtp.leap.JICP.JICPPeer@5f8ed237[Cannot bind server socket to localhost port 1099]`. Astfel, trebuie avut grijă ca la lansarea unei noi platforme să se verifice toate consolele deschise, inclusiv cele din Eclipse (Fig.2.10). Această eroare poate apărea din diferite motive (ex.: închiderea agentului RMA nu închide și platforma, console Eclipse ascunse, lansarea unei configurații suplimentare cu opțiunea -gui în loc de -container etc);

Fig.2.10 – Eroare lansare platformă multiplă în Eclipse

- **Lipsa actualizării CLASSPATH și PATH, inclusiv cu calea curentă**. Calea curentă este specificată prin . (punct);
- **Poziționarea greșită împreună cu definirea greșită a pachetului în care se găsește definiția agentului**. Ambele probleme sunt identificate ca urmare a execuției platformei (apare agentul RMA), dar fără agentul specificat. În consolă apare eroarea din figura 2.11. Trebuie avut grijă ca atunci când clasa este pusă într-un pachet, fișierul în care este implementată să se găsească într-o structură de directoare aferentă (ex.: pachetul exemple.salut presupune poziționarea fișierului sursă în directorul salut, care la rândul său este pus în directorul exemple).

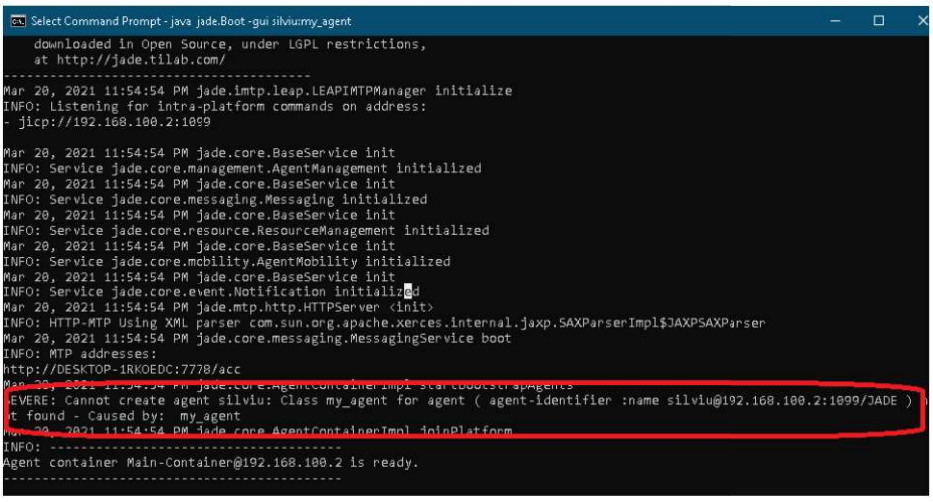

Fig.2.11 – Eroare lansare platformă multiplă în Eclipse

7. Creare de agenți JADE

Pentru crearea unui agent nou se pleacă de la clasa de bază `jade.core.Agent`, agentul nou creat moștenind funcțiile și atributele unui agent de bază. După scrierea comportamentului și a funcționării acestuia, el trebuie lansat în execuție cu sau fără parametri. Crearea unui agent JADE se realizează prin definirea unei clase care extinde clasa `jade.core.Agent` și implementarea unei metode `setup()` ca în secvența următoare:

```
import jade.core.Agent;

public class HelloWorldAgent extends Agent {

 protected void setup() {

// Printout a welcome message

System.out.println("Hello World. I'm an agent!");

  }

}
```

O clasă, cum ar fi clasa *HelloWorldAgent* prezentată anterior, reprezintă un tip agent, la fel cum o clasă Java normală reprezintă un tip obiect. La rulare pot fi lansate mai multe instanțe ale clasei *HelloWorldAgent*. Spre deosebire de obiectele Java normale, care sunt manipulate de referințele lor, un agent este întotdeauna instanțiat de instanța JADE și referința sa nu este niciodată expusă în afara agentului în sine (cu excepția cazului în care, desigur, agentul face acest lucru în mod explicit). Agenții nu interacționează niciodată prin apeluri de metode, ci prin schimbul de mesaje asincrone.

Metoda setup (Fig.2.12) include inițializările agenților. Activitatea efectivă pe care trebuie să o îndeplinească un agent se desfășoară, de obicei, în cadrul comportamentelor. Exemple de operații tipice pe care un agent le efectuează în metoda sa `setup()` sunt: afișarea unei interfețe grafice (GUI), deschiderea unei conexiuni la o bază de date, înregistrarea serviciilor pe care le furnizează în catalogul Paginilor aurii și pornirea comportamentelor inițiale. Este o bună practică să nu definim niciun constructor într-o clasă agent ci să efectuăm toate inițializările în interiorul metodei `setup()`. Acest lucru se datorează faptului că la momentul construcției agentul nu este încă legat de platforma JADE și, astfel, unele dintre metodele moștenite de la clasa `Agent` pot să nu funcționeze corect.

Un agent este definit de trei etape mari de la concepția lui și până la terminare:

- inițializare. Codul ce se dorește a fi executat la inițializarea agentului se scrie în cadrul funcției `setup()`. Aici se mai pot aduce și completări ale comportamentului inițial;
- executarea comportamentului (ciclul de viață al agentului). După cum spune și numele, aceasta este o structură ciclică în care: a) se verifică dacă a fost apelată metoda `doDelete()` a agentului: dacă a fost apelată, agentul își termină ciclul de viață, dacă nu a fost apelată, se trece la următorul comportament din lista de comportamente active; b) se execută acțiunea asociată comportamentului curent; c) se verifică terminarea comportamentului: dacă comportamentul nu s-a terminat, se trece înapoi la etapa de verificare a apelării `doDelete()`, dacă comportamentul s-a terminat, se va scoate comportamentul din lista de comportamente active și se trece înapoi în etapa de verificare a apelării `doDelete()`;
- sfârșitul rulării. Se apelează funcția `takeDown()`, în care sunt implementate operațiile de terminare asociate agentului.

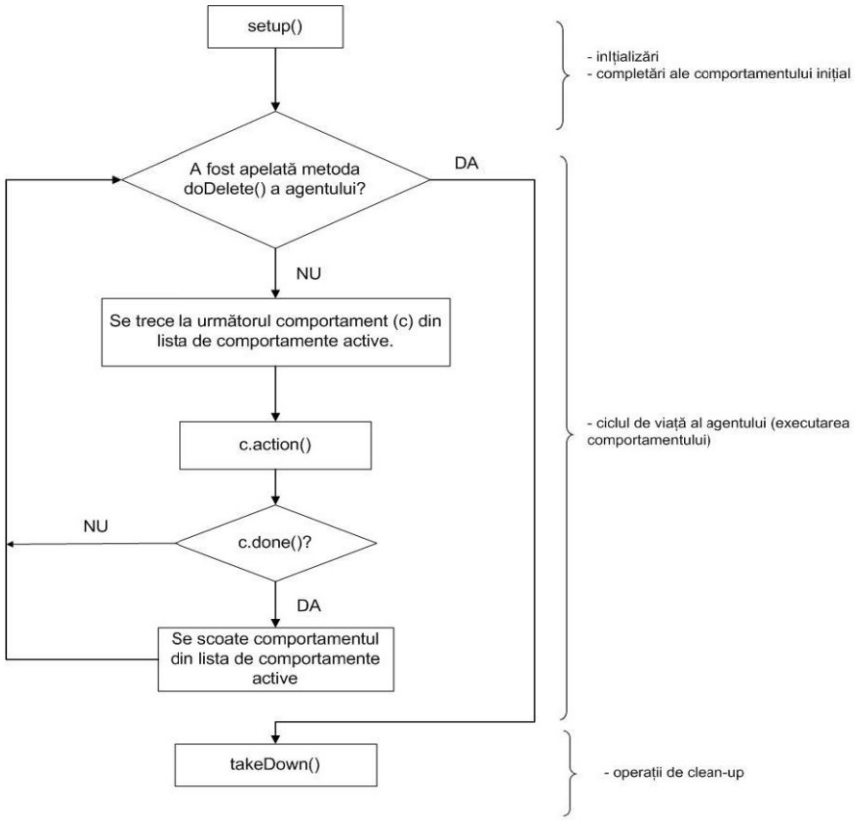

Fig.2.12 – Firul de execuție al unui agent

În JADE, ceea ce trebuie să realizeze agenții se implementează prin intermediul comportamentelor. Un comportament (behaviour) reprezintă o sarcină pe care un agent o execută și este implementat ca un obiect al unei clase care extinde `jade.core.behaviours.Behaviour`. Pentru ca agentul să execute respectivul comportament, el trebuie să îl includă în corpul clasei sale prin apelarea metodei `addBehaviour()`. Un comportament poate fi adăugat în orice moment al execuției agentului, fie din metoda de `setup()`, fie din interiorul altui comportament.

Fiecare clasă care extinde Behaviour trebuie să implementeze două metode: metoda `action()`, care definește operațiile ce vor fi realizate atunci când comportamentul este în execuție și metoda `done()`, care returnează o valoare de tip booleean pentru a indica dacă s-a realizat un comportament sau dacă el a fost retras din execuție.

a. Identificatori de agent

În conformitate cu specificațiile FIPA, fiecare instanță de agent este identificată printr-un „identificator de agent“. În JADE, un identificator de agent este reprezentat ca o instanță a clasei `jade.core.AID`. Metoda `getAID()` a clasei Agent permite regăsirea identificatorului agentului local. Un obiect AID include un nume unic global (GUID) plus un

număr de adrese. Numele în JADE are forma <*nume-local*>@<*nume-platformă*> astfel încât un agent numit Mihai, care se execută pe o platformă numită *platforma-test*, va avea ca nume global unic: Mihai@platforma-test. Adresele incluse în AID sunt adresele platformei în care agentul este executat. Aceste adrese sunt utilizate numai atunci când un agent trebuie să comunice cu un alt agent care se află pe o altă platformă compatibilă FIPA.

Clasa AID furnizează metode pentru a regăsi numele local (`getLocalName()`), GUID -ul (`getName()`) și adresele (`getAllAddresses()`). Prin urmare, putem detalia mesajul de bun venit al clasei *HelloWorldAgent*, mesaj anterior prezentat, după cum urmează:

```
protected void setup() {
 // Afiseaza mesajul de bun venit System.out.println("Hello World. I'm an
agent!"); System.out.println("My local-name is "+ getAID().getLocalName());
System.out.println("My GUID is "+ getAID().getName());
System.out.println("My addresses are:");
 Iterator it = getAID().getAllAddresses();
 while (it.hasNext()) {
System.out.println("- "+it.next());
   }
}
```

Numele local al unui agent este atribuit la momentul de pornire de către creator și trebuie să fie unic în cadrul platformei. Dacă există deja în platformă un agent cu același nume local, platforma JADE nu permite crearea noului agent. Cunoscând numele local al unui agent, AID-ul său poate fi obținut după urmează:

```
String localname = "Mihai";
AID id = new AID(localname, AID.ISLOCALNAME);
      Numele platformei este adăugat automat la GUID-ul noului creat AID de
către platforma JADE. În mod similar, cunoscând GUID-ul unui agent, AID al
său poate fi obținut după urmează:
String guid = "Mihai@ nume-platformă";
AID id = new AID(guid, AID.ISGUID);
```

b. Inițializarea agenților

Clasa HelloWorldAgent descrisă anterior poate fi compilată, ca și în cazul claselor Java normale, prin instrucțiunea următoare:

```
javac -classpath HelloWorldAgent.java
```

Pentru o compilare fără erori, bibliotecile JADE trebuie să fie în CLASSPATH. Pentru a executa un agent *HelloWorld*, adică o instanță a clasei *HelloWorldAgent*, trebuie să fie pornită platforma JADE (sau să pornească odată cu agentul) și trebuie ales un nume local pentru agentul instanțiat:

```
java -classpath <clase JADE> jade.Boot Mihai:HelloWorldAgent
```

Această comandă pornește platforma JADE și îi spune să lanseze un agent al cărui nume local este Mihai și a cărui implementare este în clasa *HelloWorldAgent*. Din nou, atât bibliotecile JADE, cât și clasa *HelloWorldAgent* trebuie să fie în CLASSPATH. Ca urmare a comenzii executate, imediat după mesajele de inițializare JADE trebuie să apară următoarele texte afișate de agentul HelloWorld:

```
Hello World. I'm an agent!
My local-name is Mihai
My GUID is Mihai@nume-platformă:1099/JADE
My addresses are:
- http://nume-platformă:7778/acc
```

Numele local al agentului este Mihai, așa este specificat în linia de comandă. Deoarece nu se specifică un nume de platformă, JADE a creat unul în mod implicit, folosind gazda locală și portul de container principal: *nume-platformă:1099/JADE*. Astfel, GUID-ul agentului este *Mihai@nume-platformă:1099/JADE*. Deși acest GUID poate arăta ca o adresă, nu este o adresă. Pentru a atribui un nume unei platforme, trebuie specificată opțiunea *-name* la lansarea containerului principal, după cum urmează:

```
java -classpath <clase JADE> jade.Boot -name nume-platformă
Mihai:HelloWorldAgent
```

Dacă repornim JADE folosind această opțiune, GUID-ul agentului Mihai va deveni *Mihai@nume-platformă*. În afară de parametrul -name, există mai multe opțiuni de configurare care pot fi specificate la pornirea platformei JADE cum ar fi -gui, care este utilizat pentru a activa interfața grafică (GUI) de administrare JADE. AID-ul agentului Mihai conține o singură adresă, deoarece există un singur MTP activ în platformă la acel moment.

Există și alte modalități de a lansa agenți, cum ar fi prin intermediul GUI de administrare, prin intermediul codului care emite o solicitare către AMS sau prin utilizarea interfeței în proces.

c. Terminarea unui agent

După afișarea mesajului de bun venit, chiar dacă nu are nimic altceva de făcut, agentul HelloWorld este încă în viață. Pentru a-l face să termine, trebuie să fie apelată metoda `doDelete()`. Similar cu metoda `setup()` care este invocată pentru a inițializa un agent, metoda `takeDown()` este invocată chiar înainte ca un agent să se încheie pentru a efectua diverse operațiuni de eliberare resurse.

d. Transmiterea argumentelor către un agent

Agenții pot lua argumente de pornire care sunt preluate, ca un vector de obiecte, prin metoda `getArguments()` a clasei `Agent`. La lansarea unui agent din linia de comandă, argumentele de pornire pot fi specificate între paranteze și separate prin virgulă (fără spații intermediare), după cum se arată în continuare:

```
java -cp jade.Boot -name nume-platformă Mihai:HelloWorldAgent
(arg1,arg2,arg3);
```

Preluarea argumentelor se face conform secțiunii de cod următoare (Fig.2.13), prin apelarea funcției getArguments care întoarce un vector de obiecte. Acestea trebuie convertite la tipul aferent argumentului specificat în linia de comandă (Fig.2.14); altfel, va rezulta o eroare iar agentul își va opri execuția pe platformă.

agent_demo.java | HelloWorldAgent.java | demo1.java | mySimpleBehaviour.java

```
1  import jade.core.Agent;
2  import jade.core.behaviours.*;
3
4  public class demo1 extends Agent {
5
6      protected void setup() {
7          Object[] args = getArguments();
8          String s;
9          int nr=0;
10         String sir="";
11
12         System.out.println("Ma cheama " + getName());
13
14         if (args != null) {
15             for (int i = 0; i < args.length; i++) {
16                 s = (String) args[i];
17                 System.out.println("p" + i + ": " + s);
18             }
19
20             // Extracting the integer.
21             nr = Integer.parseInt((String) args[0]);
22             System.out.println("i*i= " + nr * nr);
23             sir = (String) args[1];
24         }
25
26         addBehaviour(new mySimpleBehaviour(this, nr, sir));
27     }
28
29 }
30
```

Fig. 2.13 – Secvență cod pentru procesarea argumentelor din lina de comandă

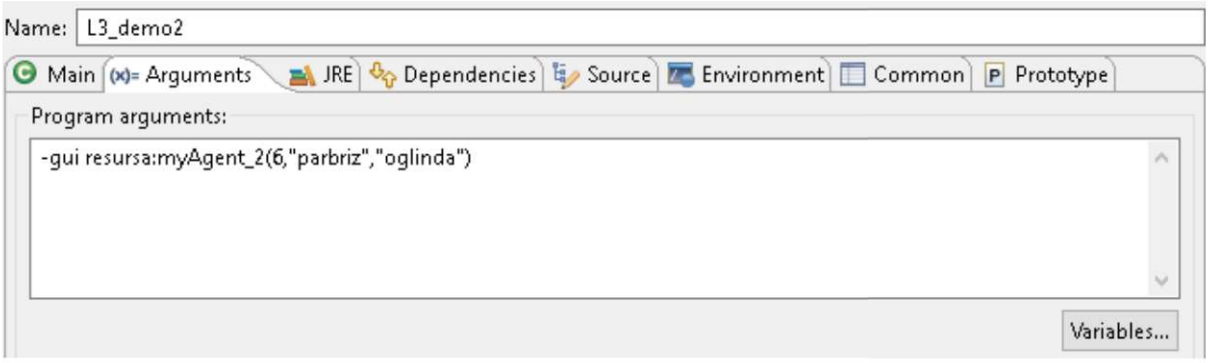

Fig.2.14 – Transmiterea parametrilor din linia de comandă din Eclipse

e. Crearea de agenți din cadrul altor agenți

În anumite situații, numărul de agenți implicați într-un proiect este variabil pe parcursul ciclului de viață; de aceea este nevoie de o metodă de a lansa în execuție agenți noi din cadrul agenților existenți. JADE oferă atât suportul pentru lansarea de agenți noi din cadrul agenților existenți, cât și suportul de a lansa agenți sau platforme complete din aplicații externe. Ceea ce suportă acest proces este o interfață care poate fi folosită pentru lansarea mediului run-time din aplicații externe. Aceasta este implementată de clasa `jade.core.Runtime`. Conform șablonului unic (*singleton pattern*), o singură instanță a acestei clase există într-o mașină virtuală, instanță care poate fi accesată prin metoda statică `instance()`. Instanța unică de Runtime oferă două metode: pentru crearea unui container principal (*createMainContainer*) și pentru crearea unui container periferic/secundar (*createAgentContainer*). Ambele metode necesită ca parametru un obiect de tip *Profile* care conține informațiile necesare execuției JADE (platformă, port ș.a.). Toate opțiunile care pot fi specificate la pornirea JADE din linia de comandă sunt disponibile drept constante în clasa *Profile* și pot fi setate folosind metoda `setParameter(String opțiune, String valoare)`.

Atât metoda `createMainContainer()`, cât și metoda *createAgentContainer()*, returnează un obiect de tip `jade.wrapper.ContainerController`. Acest obiect se ocupă cu funcționalitatea de nivel înalt a containerelor în care rulează agenții, aici regăsindu-se funcții de instalare/dezinstalare a protocoalelor de transmitere a mesajelor, gestiunea containerelor și crearea de agenți noi. Metoda `createNewAgent()` a clasei `ContainerController` returnează un obiect de tipul `AgentController`, care împachetează unele funcționalități ale agentului, păstrându-i în același timp autonomia. În special, clasa `AgentController` oferă metode cu ajutorul cărora aplicația externă poate

controla ciclul de viață al agentului încapsulat, dar ascunde referința la obiectul *Agent*, astfel încât să nu fie posibilă efectuarea de apeluri directe de metodă asupra acestuia. Este de reținut că metoda `createNewAgent()` creează instanța `Agent`, dar nu o pornește, deoarece acest lucru este posibil numai prin metoda `start()` a obiectului AgentController returnat.

În cele ce urmează este prezentată o secțiune din codul care creează o platformă multi-agent compusă din container principal, interfață grafică (RMA) și un agent de test.

```
import jade.core.Runtime;
import jade.core.Profile;
import jade.core.ProfileImpl;
import jade.wrapper.AgentController;
import jade.wrapper.ContainerController;

public class MyStandaloneApp {

	public static void main(String[] args) {
		System.out.println("aplicatie de test pentru crearea de agenti
din aplicatii externe");
		startBuyerAgent("localhost", "1099", "hydra111");
	}

	public static AgentController startBuyerAgent(String host, //
containerul principal al proiectului
			String port, // portul pe care ruleaza containerul
principal
			String name // numele agentului
	) {
		// accesul la instanta singleton runetime-ului JADE
		Runtime rt = Runtime.instance();
		// Create a container to host the Book Buyer agent
		Profile p = new ProfileImpl();
		p.setParameter(Profile.MAIN_HOST, host);
		p.setParameter(Profile.MAIN_PORT, port);
		p.setParameter(Profile.GUI, "true");
		// ContainerController cc = rt.createAgentContainer(p); - daca
se doreste doar
		// crearea agentului si atasarea lui la o platforma curenta
		ContainerController cc = rt.createMainContainer(p);
		if (cc != null) {
			// Create the Book Buyer agent and start it
			try {
				AgentController ac = cc.createNewAgent(name,
"test_external_app.MyHelloAgent", null);
				ac.start();
				return ac;
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
		return null;
	}

	}
```

Capitolul 3: Tipuri de agenți JADE

Cuprins

1. Introducere

Aplicațiile software multi-agent sunt, în general, destul de complexe. Acestea sunt adesea distribuite pe mai multe sisteme de execuție (PC-uri sau sisteme embedded) și au în componență un număr mare de procese cu mai multe fire de execuție (în cazul particular al JADE, mai multe containere cu mai mulți agenți, fiecare agent reprezentând un singur fir de execuție) și sunt dinamice prin faptul că agenții pot apărea (inițializare), dispărea (terminare) și migra între platforme. Aceste aspecte implică dificultăți în gestionare și în special în depanare. Pentru a facilita aceste procese, JADE are un serviciu de notificare a evenimentelor care stă la baza consolei de administrare JADE RMA (*Remote Management Agent*) și un set de instrumente grafice care sunt furnizate pentru a ajuta în faza de gestionare și depanare. Toate aceste instrumente sunt grupate în pachetul jadeTools.jar.

JADE este o platformă software implementată în limbajul de programare Java, care oferă un middleware pentru dezvoltarea aplicaţiilor de tip multi-agent. Arhitectura platformei JADE se bazează pe containere, ce pot fi distribuite în rețea. Containerele sunt procese Java ce conțin serviciile necesare execuției agenților. JADE oferă anumite facilități sub formă de agenți standard, precum: implementarea căutării serviciilor („pagini aurii" – DF, „pagini albe" – AMS și protocoale de interacțiune ce modelează diferite tipuri de comportamente, dintre care ne vom concentra pe cele simple.

Fiecare platformă conține un container principal numit *Main-container*, primul container lansat, iar celelalte containere trebuie să se înregistreze la el. *Main-container* are următoarele responsabilități: gestiunea tabelului de containere, care conține referințele și adresele containerelor; gestiunea tabelului de descriptori a agenţilor, tabel ce conține toți agenții prezenți pe platforma curentă, starea și locația lor, și lansarea agenților AMS (*Agent Management System*) și DF (*Directory Facilitator*) care oferă serviciile pentru gestiunea și monitorizarea agenţilor.

În continuare vor fi descriși agenții de bază care rulează odată cu platforma sau care pot fi lansați ulterior pentru a monitoriza interacțiunea între agenții componenți.

2. Agentul AMS (*Agent Management System*)

Agentul AMS coordonează şi monitorizează întreaga platformă. AMS este folosit de către agenţi pentru a căuta alţi agenţi (după nume) sau pentru a-şi gestiona ciclul de viaţă. Înregistrarea agenţilor la AMS se face automat de către platforma JADE. Agentul AMS este o componentă obligatorie a unei platforme multi-agent, fiind responsabil cu gestionarea funcționării platformei: crearea și ștergerea agenților și supravegherea migrației agenților către alte platforme și către platforma curentă. Fiecare agent trebuie să se înregistreze la un AMS pentru a obține un AID care este apoi reținut de AMS, ca o evidență a tuturor agenților pe care îi conține împreună cu starea lor actuală (de exemplu, activ, suspendat sau în așteptare). Descrierile agenților pot fi modificate ulterior sub restricție de autorizare din partea AMS. Viața unui agent cu platforma multi-agent se încheie odată cu ștergerea acestuia din AMS. După ștergere, AID-ul acelui agent poate fi eliminat de către director și poate fi pus la dispoziția altor agenți care ar trebui să îl solicite. Descrierile agenților pot fi căutate și în cadrul AMS, iar AMS este responsabil cu descrierea platformei multi-agent care poate fi preluată prin apelarea unei metode de tip get-description.

AMS poate solicita ca un agent să îndeplinească o anumită funcție de management, cum ar fi să-și încheie execuția și are autoritatea de a impune operațiunea dacă cererea este ignorată. Un singur AMS poate exista în fiecare platformă multi-agent. Dacă platforma se întinde pe mai multe mașini, AMS este autoritatea pentru toate acele mașini.

3. Agentul DF (*Directory facilitator*)

DF este o componentă opțională a unei platforme agent care furnizează servicii de tip „pagini aurii“ altor agenți (căutare agenți după serviciile pe care le oferă). Acesta deține o listă actualizată a agenților și trebuie să furnizeze cele mai actuale informații despre ei. Fiecare agent care dorește să își facă publice serviciile altor agenți trebuie să găsească un DF adecvat și să solicite înregistrarea descrierii sale, care poate sau nu să fie acceptată de DF. Agenții pot solicita ulterior retragerea unei descrieri, moment în care nu mai există o legătură cu DF pentru a intermedia informații referitoare la agentul respectiv sau pot solicita modificarea descrierii. În plus, un agent poate emite o cerere de căutare către un DF pentru a găsi descrieri care corespund unor criterii anumite de căutare (ex.: nume serviciu). DF nu garantează valabilitatea informațiilor furnizate ca răspuns la o cerere de căutare (ex.: un agent se poate înregistra cu un serviciu, apoi, ca urmare a unei defecțiuni își oprește execuția, dar serviciul rămâne înregistrat pe DF), însă poate restricționa accesul la informațiile din directorul său și va verifica toate permisiunile de acces pentru agenții care încearcă să-l informeze cu privire la modificările stării agentului.

4. Agentul RMA (*Remote Monitoring Agent*)

JADE RMA (*Remote Monitoring Agent*) este un instrument de sistem implementat de clasa `jade.tools.rma.rma` care construiește o consolă grafică de gestionare a platformei. Se poate lansa din linia de comandă folosind opțiunea “-gui” din linia de comandă. Acesta oferă o interfață vizuală pentru administrarea unei platforme JADE distribuite, ce se compune dintr-una sau mai multe gazde și noduri container. Pot fi lansate și alte instrumente pornind de la RMA. Mai multe RMA pot fi lansate pe aceeași platformă, fiecare cu un nume diferit.

Când este pornit, agentul RMA se înregistrează la AMS pentru a fi notificat cu privire la toate evenimentele la nivel de platformă. Platforma se poate asemăna cu un copac de containere ale căror frunze sunt agenții. Acest panou este implementat de clasa jade.gui.AgentTree și este reutilizat de majoritatea celorlalte instrumente. Există trei tipuri de noduri: platformă, container și agent. Dacă este selectat un agent, meniul “*pop-up*” permite agentului să fie suspendat, reluat, terminat, clonat, salvat, înghețat sau migrat într-un alt container (Fig.3.1). De asemenea, permite asamblarea și trimiterea unui mesaj personalizat, ad-hoc. Dacă este selectat un container, meniul pop-up permite crearea unui agent nou, încărcarea unui agent existent, instalarea sau îndepărtarea unui MTP, salvarea sau încărcarea

containerului, inclusiv a tuturor agenților acestuia și terminarea containerului. Dacă este selectată o platformă, meniul pop-up permite vizualizarea profilului platformei. Meniul permite, de asemenea, gestionarea MTF-urilor platformei.

RMA poate fi utilizată pentru a controla un set de platforme, cu condiția ca toate acestea să fie conforme cu standardul FIPA (www.fipa.org). Pot fi utilizate numai acele mesaje și acțiuni de gestionare definite de FIPA, în loc de cele disponibile prin JADE IMTP în cadrul oricărei platforme JADE unice. Pentru a comunica cu o platformă la distanță, trebuie furnizat identificatorul AMS (și anume, AMS AID), care trebuie să includă numele acesteia și cel puțin o adresă de transport valabilă.

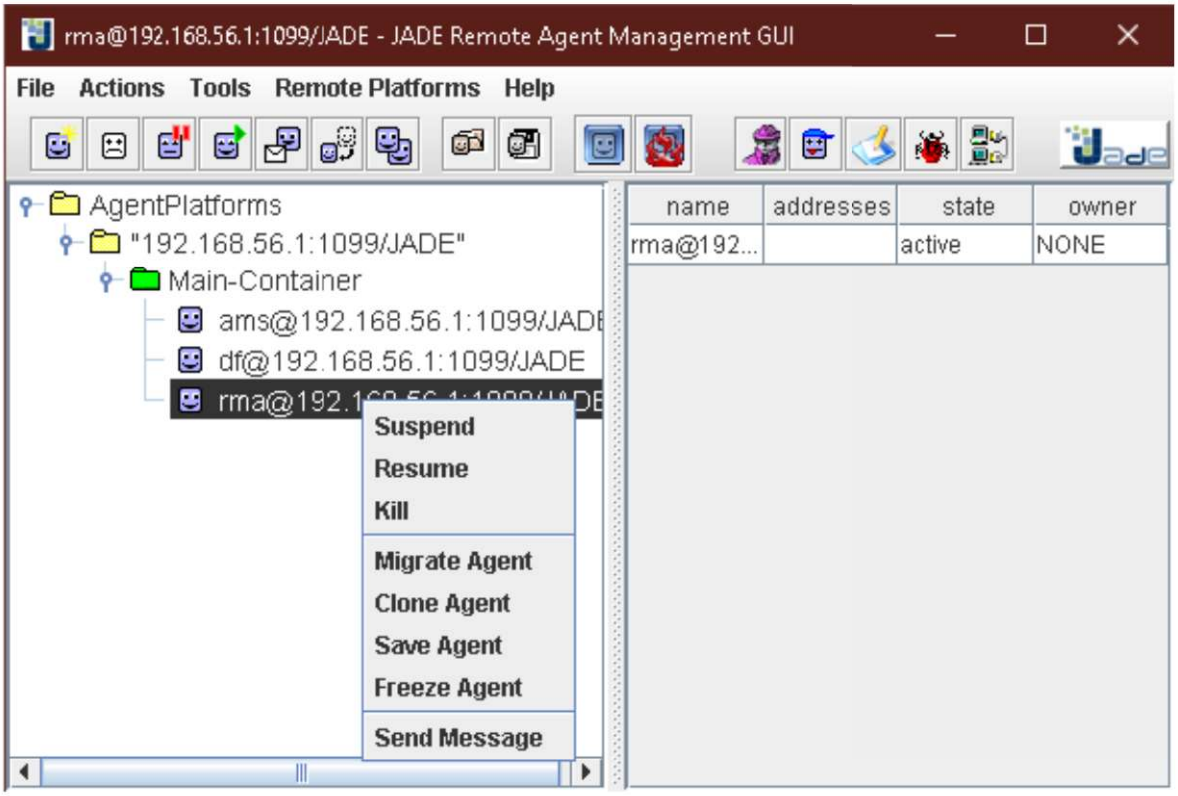

Fig.3.1 – Interfața grafică a agentului de monitorizare RMA

Din interfața grafică a agentului RMA pot fi lansați în execuție agenți din calea curentă de execuție sau agenți platformă (ex.: un nou agent RMA). În cazul platformelor care rulează pe mai multe sisteme de calcul, agentul RMA trebuie pornit cu specificarea locației în care se află implementarea agenților ce trebuie lansați. În figura 3.2 este exemplificat modul de creare a unui agent nou (Actions->Start New Agent). Trebuie avut în vedere să se selecteze containerul pe care să ruleze noul agent și tipul lui. Toate tipurile de agenți existenți în lista de selecție sunt opțiuni valide, singura restricție de creare fiind ca instanța agentului să aibă un nume unic. Adițional fi pot specificați și parametrii de rulare ai agentului.

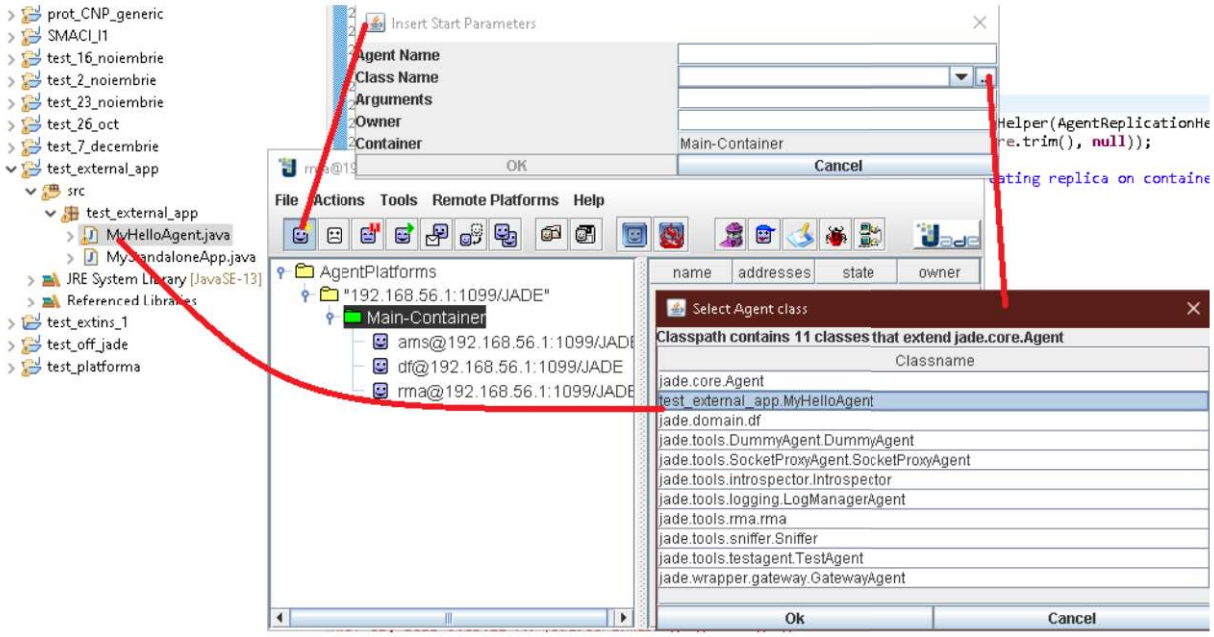

Fig.3.2 – Interfața grafică a agentului de monitorizare RMA

5. Agentul Dummy

Agentul Dummy (*Dummy Agent*) este un instrument utilizat pentru trimiterea de mesaje ACL personalizate, ca să se testeze comportamentul unui alt agent sau al unui protocol de interacțiune. Acesta este implementat în clasa `jade.tools.DummyAgent.DummyAgent`, putând fi lansat similar unui agent comun (Fig.3.2). DA are funcția de a trimite și primi mesaje personalizate care pot fi compuse folosind o interfață grafică simplă (Fig.3.3). Alternativ, mesajele pot fi încărcate ori salvate dintr-un fișier conform șablonului prezentat în continuare. Datorită simplității și eficienței lui, este mult utilizat în testarea protocoalelor de interacțiune (schimb structurat de mesaje). Mai multe instanțe ale DummyAgent pot fi lansate acolo unde este necesar, atât din meniul RMA, cât și din linia de comandă:

```
prompt>java jade.Boot myDummy:jade.tools.DummyAgent.DummyAgent
```

Șablon mesaj agent:

```
(ACCEPT-PROPOSAL
 :sender   ( agent-identifier :name aaaaa@192.168.56.1:1099/JADE  :addresses
(sequence http://DESKTOP-1RKOEDC:7778/acc ))
 :receiver   (set ( agent-identifier :name da111@192.168.56.1:1099/JADE ) )
 :content   "continut mesaj"
)
```

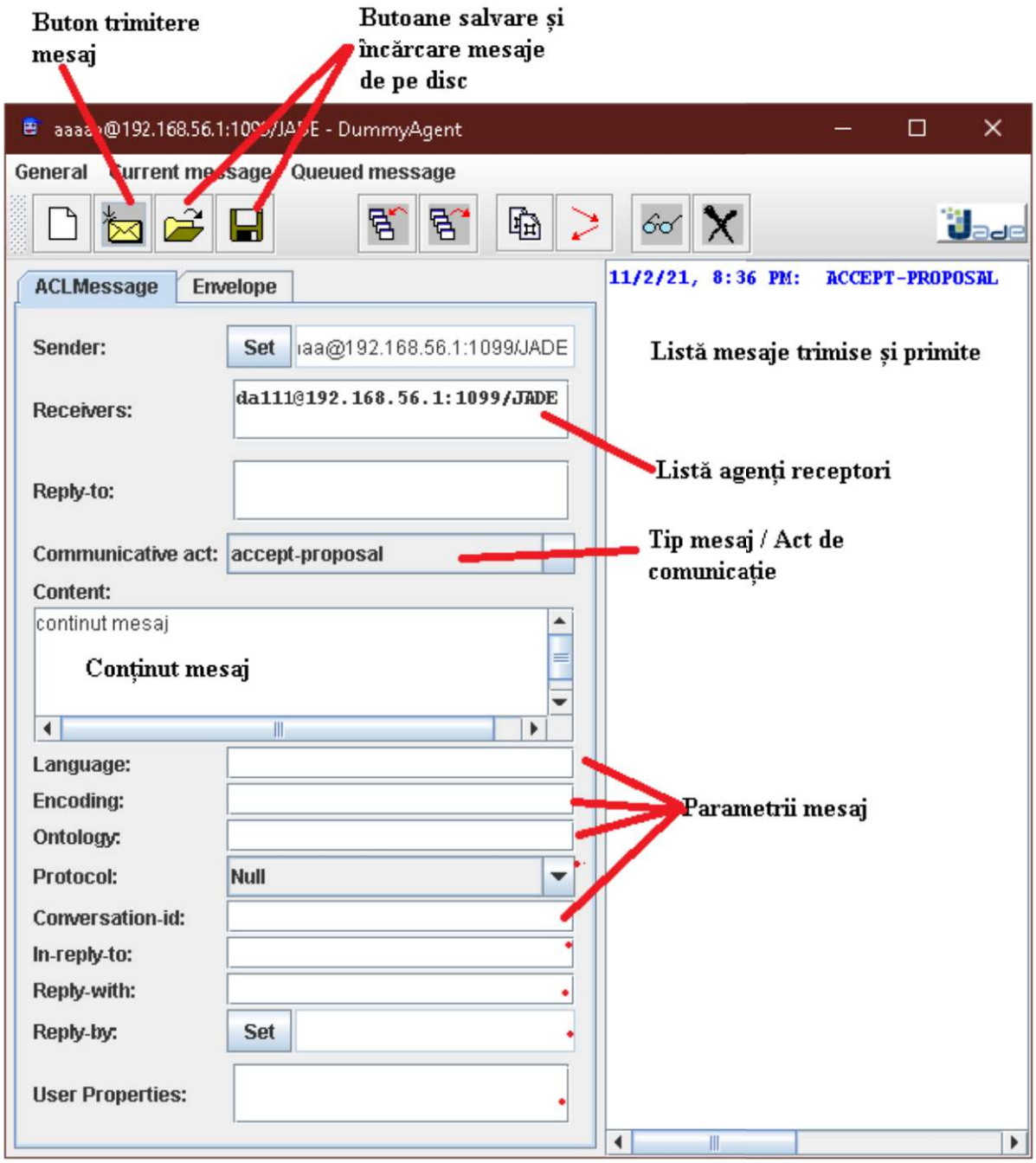

Fig.3.3 – Interfața grafică a agentului DummyAgent

6. Agentul Sniffer

După cum îi spune și numele, acest agent investighează și se documentează cu privire la conversațiile altor agenți, dar este utilizat și pentru depanare. Acest agent este implementat de clasa `jade.tools.sniffer.Sniffer`. Agentul Sniffer se abonează la o platformă AMS ca să fie notificat cu privire la toate evenimentele de pe platformă și la toate schimburile de mesaje dintre un set de agenți specificați. Când utilizatorul decide să monitorizeze un agent

sau un grup de agenți, fiecare mesaj direcționat către sau provenind de la ei este urmărit și afișat în interfața grafică a agentului *Sniffer*. În cazul platformelor tolerante la defect, serviciul de notificare trebuie pornit în mod explicit.

Agentul *Sniffer* folosește aceeași intrefață ca și agentul RMA, dar acesta e folosit pentru navigarea pe platforma agentului și selectarea agenților a căror conversație urmează să fie monitorizată.

Utilizatorul poate selecta și vizualiza detaliile fiecărui mesaj individual, poate salva mesajul pe disc ca fișier text sau poate converti în binar o conversație întreagă . Mai multe instanțe ale acestui agent pot fi lansate pe orice container unic din meniul de instrumente al RMA și din linia de comandă:

```
prompt> java jade.Boot mySniffer:jade.tools.sniffer.Sniffer
```

Dacă există fișierul ”sniffer.inf” în directorul de lucru curent, el va fi citit de agentul *Sniffer* la momentul lansării, și considerat ca o listă de agenți de investigat. Fiecare linie din fișier conține în mod obligatoriu un nume de agent și, eventual, o listă de acțiuni (INFORM, PROPOSE etc – tipuri de mesaje ACL – *Agent Communicative Language*). De exemplu, linia:

```
ams inform propose m*
```

îi spune agentului Sniffer să testeze următorii agenți: platforma AMS (selectând numai acele mesaje ACL al căror tip este INFORM sau PROPOSE) și orice agent al cărui nume începe cu litera „m“. O listă de nume de agenți poate fi, de asemenea, transmisă ca argument în linia de comandă la lista agenților ce urmează a fi investigați de îndată ce apar pe platformă.

În cele ce urmează este prezentată modalitatea de monitorizare a unei conversații între doi agenți de tipul *DymmyAgent*. Astfel, se va selecta containerul principal în care se pornesc agenții *DummyAgent* (X2) și Sniffer. În primul pas trebuie configurat din Sniffer care sunt agenții monitorizați (Fig.3.4) și adăugați la diagrama secvențială a schimbului de mesaje. Ulterior, se asamblează un mesaj într-unul din agenții Dummy pentru a fi trimis celuilalt agent Dummy. Mesajul trimis poate fi vizualizat în diagrama secvențială și poate fi inspectat prin selectare cu click dreapta și apoi *View*. Structura mesajului este prezentată în partea dreaptă a Fig3.4. Pot fi vizualizate doar conversațiile în care cel puțin unul din participanți apare în lista de monitorizare în momentul schimbului de mesaje, ulterior putând fi selectat și al doilea participant. Dacă schimbul de mesaje are loc fără ca unul din agenți să fie monitorizat, atunci informația respectivă este pierdută.

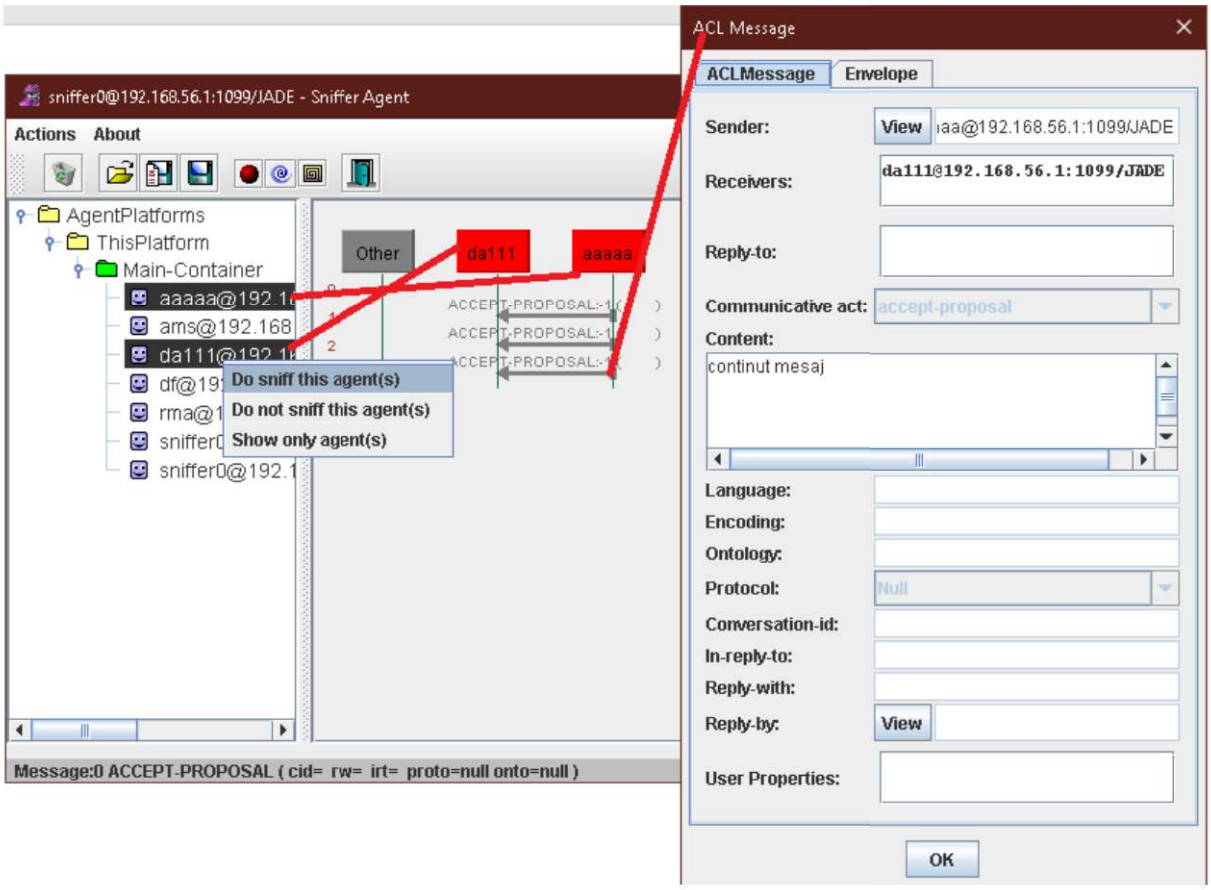

Fig.3.4 – Interfața grafică a agentului *Sniffer*
și modalitatea de monitorizare a agenților implicați într-o interacțiune

7. Agentul Introspector

Agentul Introspector este folosit pentru a depana comportamentul unui singur agent (analizează execuția unui agent). Acest instrument permite monitorizarea și controlarea ciclului de viață al unui agent și al cozilor sale de mesaje trimise și primite. De asemenea, monitorizează reacțiile la stimuli externi, analizează care comportamente sunt executate și care trec în coada de comportamente programate, inclusiv capacitatea utilă de a executa comportamente pas cu pas. Acest agent se pornește din RMA și ulterior sunt selectați agenții care urmează să fie monitorizați (Fig.3.5). Este posibil să se monitorizeze starea agentului (activ, suspendat, inactiv, în așteptare, în curs de mutare, oprit), comportamentele existente, mesajele primite și cele recepționate.

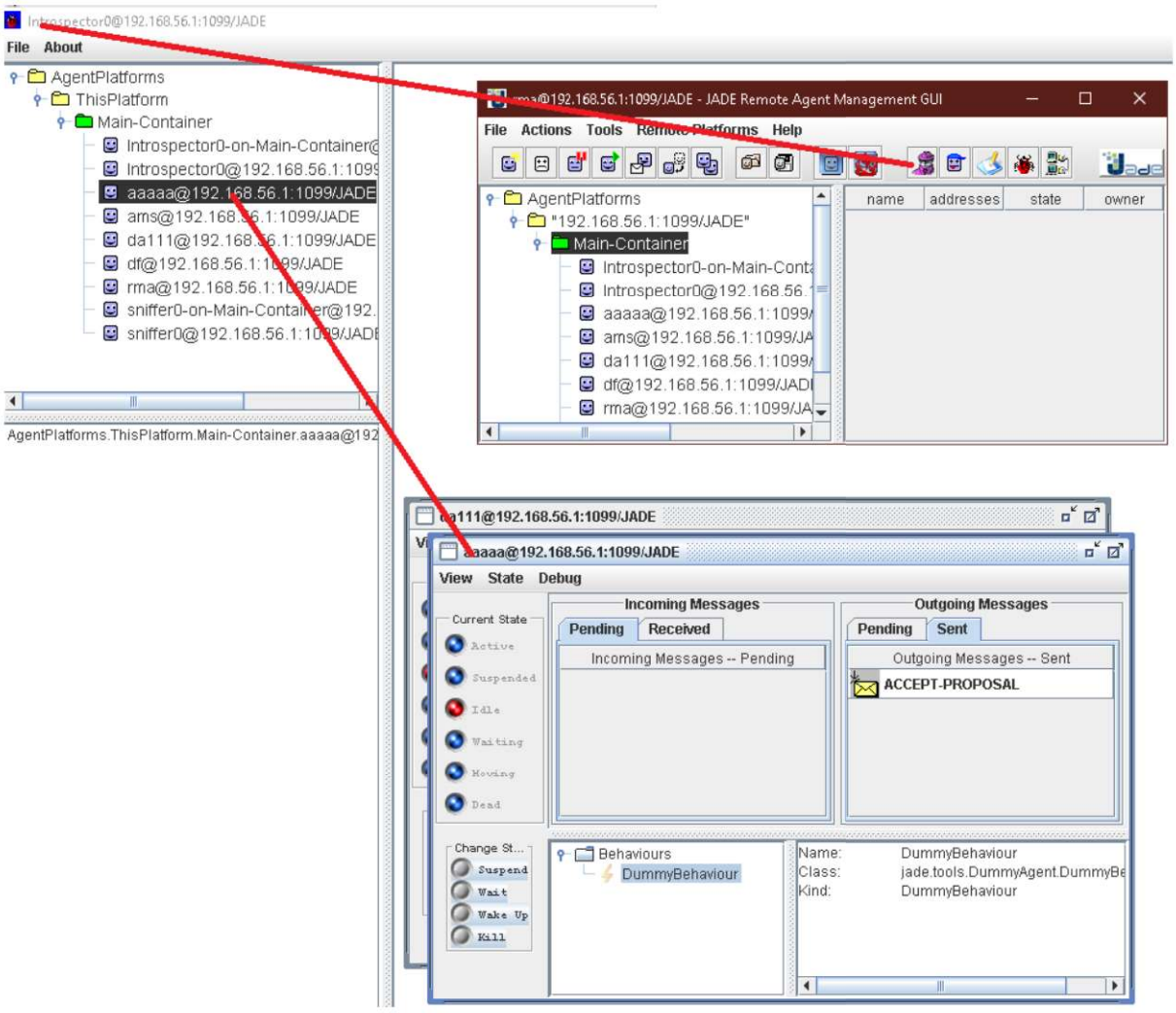

Fig.3.4 – Interfața grafică a agentului Introspector

8. Agentul de înregistrare a acțiunilor (*Log Manager*)

Agentul Log Manager este un instrument care simplifică gestionarea dinamică și distribuită prin furnizarea unei interfețe grafice care permite modificarea nivelurilor de logare ale fiecărei componente a platformei JADE în timpul rulării platformei. Aceasta include toate acele componente care sunt executate pe noduri distante, inclusiv mesajele de înregistrare specifice aplicației.

Managerul de jurnale exploatează capacitățile API-urilor java.util.logging pe care se bazează înregistrarea în jurnal JADE. Fiecare clasă utilizează propria instanță de *Logger* numită după numele de clasă.

Obiectul *Logger* poate fi configurat cu propriul nivel de înregistrare în jurnal și propriul set de handlere. Această configurație poate fi statică – specificând un fișier de configurare

`java.util.logging` la momentul de lansare – sau dinamică, utilizând agentul Log Manager. De exemplu, următoarea linie de comandă lansează un container JADE și specifică un fișier de configurare pentru a inițializa sistemul de înregistrare în jurnal al JVM:

```
prompt> java -
Djava.util.logging.config.file=logging.properties
jade.Boot -container
```

Fișierul de configurare a înregistrării în jurnal este în format java.util.Properties standard. În cazul în care conținutul fișierului *logging.properties* este după cum urmează:

```
handlers = java.util.logging.ConsoleHandler
level = OFF
jade.core.messaging.level = FINEST
```

containerul JADE lansat va înregistra numai mesajele jurnal ale subsistemului *Messaging*, adică din toate clasele din pachetul jade.core.messaging.

Prin utilizarea consolei RMA, o instanță a unui agent Log Manager poate fi lansată de la distanță pe orice container al platformei. Agentul Log Manager poate fi utilizat, de asemenea, pentru a gestiona instrumente de înregistrare (*loggers*) specifice aplicației.

Capitolul 4: Tipuri de comportamente simple

Cuprins

1. Introducere

Un comportament reprezintă o sarcină pe care un agent o poate îndeplini și este implementat ca obiect al unei clase care extinde `jade.core.behaviors.Behaviours`. Pentru ca un agent să execute sarcina pusă în aplicare de un obiect de tip comportament, acesta din urmă trebuie adăugat agentului prin metoda `addBehaviour()` din clasa Agent. Comportamentele pot fi adăugate în orice moment atât din metoda `setup()` cât și din interiorul altor comportamente, cu condiția ca acestea din urmă să fie programate la execuție.

Sarcina (sau sarcinile reale) pe care trebuie să le facă un agent sunt realizate prin „comportamente“. Un comportament reprezintă o sarcină pe care o poate îndeplini un agent și este implementată printr-un obiect din clasa jade.core.behaviours.Behaviour. Pentru ca un agent să execute sarcina implementată de un obiect de tip comportament, comportamentul

trebuie adăugat la agent prin intermediul metodei addBehaviour() a clasei Agent. Comportamentele pot fi adăugate în orice moment după pornirea agentului (în metoda setup()) sau din cadrul altor comportamente.

Fiecare clasă care extinde comportamentul simplu trebuie să implementeze două metode abstracte. Metoda `action()` definește operațiunile ce trebuie efectuate atunci când comportamentul este în execuție. Metoda `done()` returnează o valoare booleană pentru a indica dacă un comportament s-a finalizat sau nu și urmează să fie eliminat din grupul de comportamente pe care un agent le execută.

2. Programarea și execuția comportamentelor

Un agent trebuie să poată îndeplini mai multe sarcini concurente ca răspuns la diferite evenimente externe. Pentru a eficientiza managementul agenților, fiecare agent JADE este compus dintr-un singur fir de execuție și toate sarcinile sale sunt modelate și pot fi implementate ca obiecte de tip comportament (*Behavior*). Agenții cu mai multe fire de execuție pot fi, de asemenea, implementați, dar JADE nu oferă niciun suport specific (cu excepția sincronizării cozii de mesaje ACL).

Dezvoltatorul care dorește să implementeze o sarcină specifică unui agent trebuie să definească una sau mai multe subclase de comportament, să le instanțieze și să adauge obiectele de tip comportament la lista de activități ale agentului. Clasa Agent, care trebuie extinsă de dezvoltator, expune două metode: `addBehaviour(Behaviour)` și `removeBehaviour(Behaviour)`, care permit gestionarea cozii de comportamente a unui anumit agent. De observat faptul că, comportamentele și subcomportamentele pot fi adăugate oricând este necesar, nu doar în cadrul metodei Agent.setup(), care este punctul de intrare în agentul dorit. Adăugarea unui comportament trebuie să fie văzută ca o modalitate de a genera un nou fir de execuție (cooperativ) în cadrul agent.

Un planificator, implementat de clasa agent de bază și ascuns programatorului, efectuează o politică de planificare nepreemptivă de tip circular și cu cuante de timp proporționale (*round-robin*) printre toate comportamentele disponibile în coada pregătită, executând o clasă derivată din comportament până când va elibera controlul (acest lucru se întâmplă când metoda action() returnează). Dacă sarcina de realizat nu a fost încă finalizată (`done()` returnează un rezultat diferit de adevărat), aceasta va fi reprogramată în runda următoare. Un comportament se poate bloca, de asemenea, așteptând sosirea unui eveniment

(ex.: mesaj sau semnalizare explicită). În particular, planificatorul agent execută metoda `action()` pentru fiecare comportament prezent în coada de comportamente valide; când `action()` revine, metoda `done()` este apelată pentru a verifica dacă comportamentul și-a finalizat sarcina. În caz afirmativ obiectul comportament este eliminat din coadă.

Comportamentele funcționează la fel ca un set de fire de execuție cooperative, dar nu există nicio stivă comună cu date de salvat. Prin urmare, întreaga stare a agentului gazdă trebuie memorată în variabilele de instanță ale comportamentelor și agentului asociat.

Pentru a evita o așteptare activă pentru mesaje (și, în consecință, o pierdere de timp CPU) (exemplu: while(true){așteaptă intrare}), fiecărui comportament i se permite să-și blocheze/suspende execuția. Metoda `block()` pune comportamentul într-o coadă de comportamente blocate imediat ce metoda action() returnează. Este de observat aici că efectul de blocare nu este obținut imediat după apelarea metodei `block()`, ci imediat după revenirea de la metoda `action()`. Toate comportamentele blocate sunt reprogramate de îndată ce sosește un nou mesaj, prin urmare programatorul trebuie să aibă grijă să blocheze din nou un comportament dacă mesajul primit nu a fost de interes. Mai mult, un obiect de tipul comportament se poate bloca pe sine pentru o perioadă definită de timp prin apelarea metodei block() cu o valoare ce reprezintă timpul respectiv, exprimat în milisecunde.

Datorită modelului multitasking non-preemptiv ales pentru comportamentele agenților, programatorul responsabil cu dezvoltarea proiectului multi-agent trebuie să evite să folosească bucle infinite și chiar să efectueze operațiuni de durată în cadrul metodelor action(). Atunci când se execută metoda `action()` a unui comportament, niciun alt comportament nu poate continua până la sfârșitul metodei (desigur, acest lucru este valabil numai în ceea ce privește comportamentele aceluiași agent; comportamentele altor agenți rulează în fire Java diferite și pot fi executate în continuare independent).

În plus, deoarece nu este salvat niciun context al stivei de memorie, de fiecare dată când metoda `action()` este rulată de la început nu există nicio modalitate de a întrerupe un comportament în mijlocul metodei `action()`, de a ceda CPU-ului altor comportamente și apoi de a începe comportamentul original de unde s-a realizat întreruperea. În cele ce urmează este dat un exemplu în care o sarcină care ar ocupa procesorul o durată mai îndelungată este realizată în etape în cadrul unui comportament simplu. Etapele sunt simulate prin execuția secvențială a diferitelor secțiuni de cod și avansarea automată la următoarea secțiune în etapa următoare planificată. Astfel, sarcina op() este împărțită în subsarcinile op1(), op2() și op3().

Pentru a se obține funcționalitatea dorită se apelează op1() la prima rulare, op2() la a doua rulare și op3() la a treia rulare, pentru ca apoi comportamentul să fie marcat ca terminat.

```
public class my3StepBehaviour {
      private int state = 1;
      private boolean finished = false;
      public void action() {
            switch (state) {
                  case 1: { op1(); state++; break; }
                  case 2: { op2(); state++; break; }
                  case 3: { op3(); state=1; finished = true; break; }
            }
}
public boolean done() {
return finished;}}
```

Exemplul anterior ilustrează modul de reprezentare a comportamentelor ca niște elemente de tip FSM (*finite state machines* – mașină cu stări finite), în care starea internă este memorată sub formă de variabile interne. Atunci când se operează cu comportamente agent complexe (ex.: implementarea de protocoale de interacțiune), utilizarea variabilelor de stare explicite poate fi dificilă; așa că JADE oferă un set de comportamente compozite pentru a facilita simularea mașinilor cu stări finite pornind de la comportamente simple.

Cadrul de lucru JADE oferă subclase de tip `Behaviour` care conțin sub-comportamente ce pot fi executate conform cu diferite politici. De exemplu, clasa `SequentialBehaviour` este compusă din subcomportamente care se execută secvențial pentru fiecare apelare a metodei `action()`.

În figura 4.1 sunt enumerate cele mai importante comportamente JADE, derivate din clasa de bază `Behaviour`. Pornind de la clasa de bază `Behaviour`, o ierarhie de clase este definită în pachetul `jade.core.behaviours` al distribuției de bază JADE (Fig.4.1).

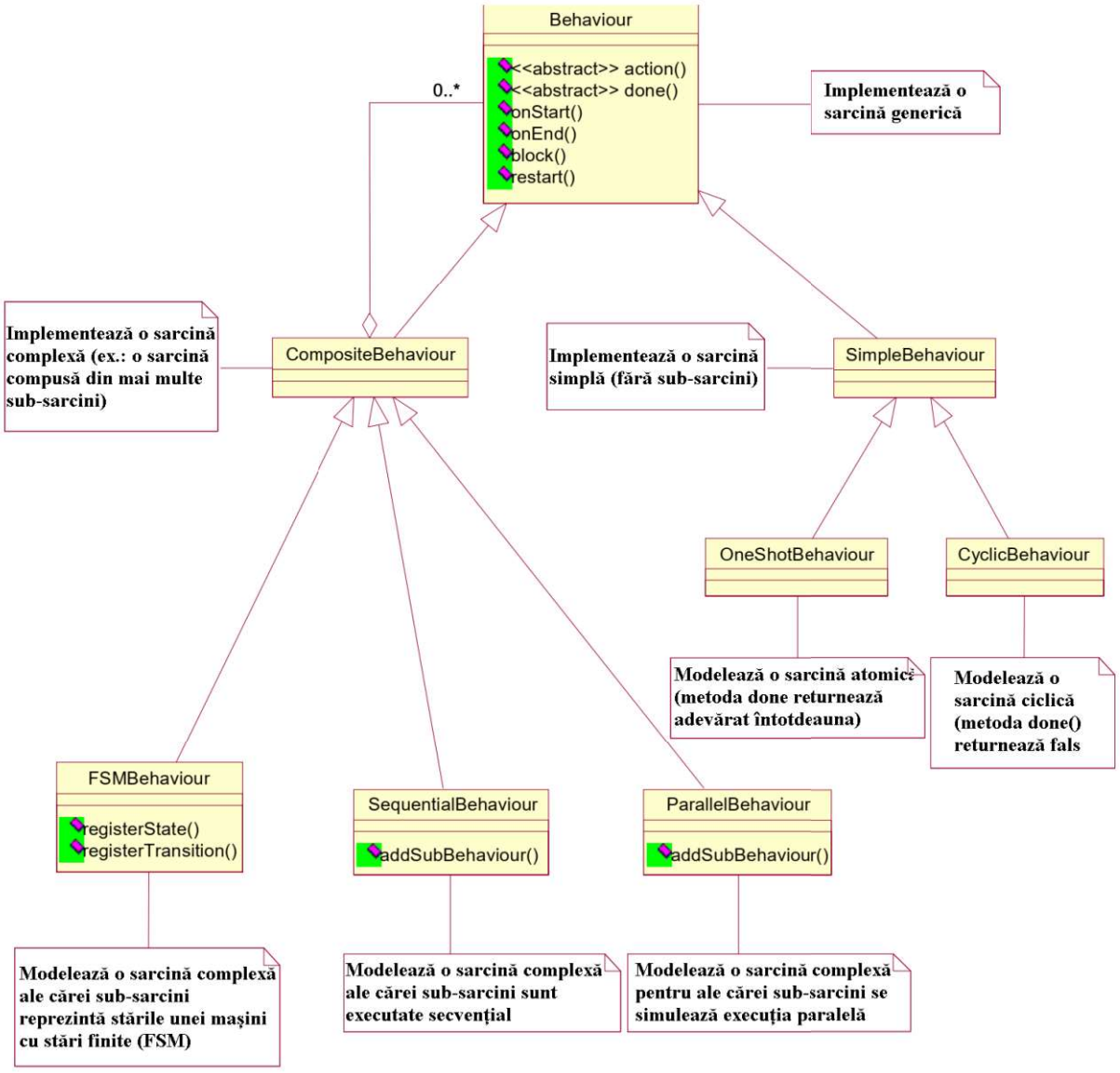

Fig.4.1 – Modelul UML al ierarhiei clasei `Behaviour`

Un agent poate executa mai multe comportamente simultan. Atunci când un comportament este programat pentru executare, metoda sa action() este apelată și se execută până când revine. Prin urmare, programatorul este cel care decide atunci când un agent trece de la executarea unui comportament la executarea altuia. Această abordare creează adesea dificultăți pentru dezvoltatorii JADE fără experiență și trebuie să fie întotdeauna reținută atunci când se scriu agenți JADE. Deși necesită un efort suplimentar, acest model de lucru are câteva avantaje:

- se permite un singur fir de execuție per agent, lucru important pentru mediile de lucru de tip embedded unde resursele sunt limitate;

- oferă performanțe îmbunătățite, deoarece comutarea comportamentului este mult mai rapidă decât comutarea firelor de execuție Java;
- elimină toate problemele de sincronizare între comportamentele concurente care accesează aceleași resurse, deoarece toate comportamentele sunt executate de același fir de execuție Java. Acest lucru are ca rezultat, de asemenea, o îmbunătățire a performanței;
- când are loc o schimbare de comportament, starea unui agent nu include nicio informație din stiva de memorie, ceea ce implică faptul că este posibil să se copieze o instanță a agentului cu tot cu informațiile aferente. Acest lucru permite implementarea unor funcții avansate importante, cum ar fi salvarea stării unui agent într-o înregistrare persistentă pentru refolosirea sa ulterioară (persistența agentului) sau transferul agentului într-un alt container pentru execuție de la distanță (mobilitatea agentului).

În JADE există un singur fir de execuție Java per agent. Deoarece agenții JADE sunt implementați în Java, totuși, programatorii pot începe noi fire de execuție Java în orice moment, după cum au nevoie. În acest caz, trebuie luat în considerare faptul că avantajele discutate anterior nu mai sunt valabile.

Succesiunea de operații din cadrul firului de execuție al unui agent este descrisă în figura 2.12. Este important de menționat (așa cum se observă și din reprezentarea grafică) faptul că execuția unui comportament va bloca execuția oricărui alt comportament atâta timp cât metoda action a comportamentului în curs de execuție nu se definitivează (returnează true, false sau altă valoare – metoda este de tipul void). Când nu există comportamente disponibile pentru execuție, firul de execuție al agentului intră în stare de repaus (*sleep state*) pentru a nu consuma timp CPU. Firul este trezit din nou odată ce un comportament devine din nou disponibil pentru execuție.

Toate comportamentele moștenesc metodele `onStart()` și `onEnd()` din clasa Behaviour. Aceste metode sunt executate o singură dată chiar înainte de primul apel la metoda `action()` și imediat după ce metoda done() returnează "true". Acestea sunt destinate să efectueze operațiuni de inițializare și terminare specifice activității. Spre deosebire de metodele `action()` și `done()` care sunt declarate abstracte, acestea au o implementare implicită goală care permite dezvoltatorilor să le reimplementeze dacă este necasar în cadrul proiectului.

Un comportament poate fi anulat în orice moment apelând metoda `removeBehaviour()` a clasei Agent. Un apel la această metodă elimină comportamentul menționat din grupul de comportamente executate în prezent de agent. În consecință, dacă un comportament este abandonat utilizând metoda `removeBehaviour()`, metoda sa `onEnd()` nu este apelată.

Fiecare comportament are o variabilă de membru numită `myAgent` care indică agentul în care execută comportamentul. Acest lucru oferă o modalitate ușoară de a accesa resursele unui agent din interiorul comportamentului (ex.: un mesaj ajunge în coada agentului și este citit printr-un comportament; astfel, este nevoie ca în comportamentul care citește coada agentului să existe o referință la părintele său – în acest caz, agentul receptor). Odată ce un comportament a fost executat, dacă trebuie executat a doua oară, este necesar să se apeleze mai întâi metoda `reset()`.

3. Comportament generic – Behaviour

Această clasă abstractă oferă un model de bază pentru implementarea sarcinilor agentului și stabilește metoda de bază pentru programarea comportamentului implementând toate tranzițiile de stare (adică pornirea, blocarea și repornirea unui obiect de tip comportament Java). Metoda `block()` permite blocarea unui comportament până când are loc un eveniment (de obicei, până la sosirea unui mesaj). Această metodă lasă neafectate celelalte comportamente ale unui agent, permițând astfel un control mai fin asupra multitasking-ului agentului. Această metodă pune comportamentul într-o coadă de comportamente blocate și are efect de îndată ce metoda `action()` își termină execuția. Toate comportamentele blocate sunt reprogramate imediat ce sosește un nou mesaj. Mai mult, un obiect de tip comportament se poate bloca pe el însuși pentru o perioadă de timp specificată, trecând o valoare de așteptare (în milisecunde) la metoda block (valoare milisecunde). Un comportament poate fi repornit în mod explicit apelând metoda `restart()`.

Rezumând, un comportament blocat își poate relua execuția într-unul din următoarele trei cazuri:

- un mesaj ACL este primit de către agentul căruia îi aparține acest comportament;
- un timeout asociat cu acest comportament de către un apel anterior block() expiră;
- metoda `restart()` este invocată în mod explicit pe acest comportament.

Clasa Behavior oferă, de asemenea, două metode virtuale, numite `onStart()` și `onEnd()`. Aceste metode pot fi suprascrise de utilizator atunci când se dorește ca unele acțiuni să fie executate înainte și după rularea execuției comportamentului (metoda `action()`). Metoda (procedura) `onEnd()` returnează un întreg care reprezintă valoarea de terminare a comportamentului. Această valoare este folosită în comportamentele complexe pentru a avea o informație despre starea care îi urmează stării curente.

Trebuie remarcat faptul că metoda `onEnd()` este apelată după finalizarea comportamentului și eliminarea acestuia din grupul de comportamente al agenților. Prin urmare, apelarea `reset()` în interiorul `onEnd()` nu este suficientă pentru a repeta ciclic sarcina reprezentată de acel comportament; în plus, comportamentul trebuie adăugat din nou agentului ca în exemplul următor:

```
public int onEnd() {
reset();
myAgent.addBehaviour(this);
return 0;
}
```

Această clasă oferă o serie de metode pentru a accesa și modifica obiecte de tip `DataStore` (stocare informații la nivel de comportament și/sau agent). `DataStore` poate fi un depozit util pentru schimbul de date între comportamente, așa cum se face, de exemplu, în clasele `jade.proto.AchieveREInitiator/Responder`. De reținut că obiectele de tip `DataStore` sunt golite și toate datele înmagazinate se pierd atunci când comportamentul este resetat.

4. Comportament simplu – *SimpleBehaviour*

Această clasă abstractă modelează comportamente atomice simple. Metoda sa `reset()` nu face nimic în mod implicit, dar poate fi suprascrisă de subclasele definite de utilizator.

5. Comportament de tip *one-shot*

Comportamentele *one-shot* sunt concepute pentru a fi finalizate într-o singură fază de execuție; metoda lor `action()` este astfel executată o singură dată. Clasa

`Jade.core.behaviors.OneShotBehaviour` are preimplementată metoda `done()`, care returnează în mod implicit valoarea adevărat, lucru care asigură neplanificarea comportamentului a doua oară. Metoda `done()` aferentă comportamentului de tip *one-shot* poate fi extinsă pentru a implementa noi tipuri de comportamente de tip one-shot.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
public void action() {
// perform operation X
}
}
```

În acest exemplu, sarcina X se efectuează o singură dată.

6. Comportament ciclic

Comportamentele ciclice sunt concepute pentru a nu se finaliza niciodată; metoda lor `action()` execută aceleași operațiuni de fiecare dată când este apelată. Clasa `jade.core.behaviors.CyclicBehaviour` implementează deja metoda `done()` astfel încât ea să returneze fals întotdeauna, lucru care asigură planificarea continuă a comportamentului. Metoda `done()` aferentă comportamentului de tip ciclic poate fi extinsă pentru a implementa noi tipuri de comportamente de tip ciclic.

```
public class MyCyclicBehaviour extends CyclicBehaviour {
public void action() {
// perform operation Y
}
}
```

În acest exemplu, operațiunea Y se efectuează repetitiv până când agentul care execută finalizează procesul.

7. Comportament de tip alarmă (*Waker*)

JADE oferă două clase gata implementate (în pachetul `jade.core.behaviors`) care pot fi folosite pentru a produce comportamente ce se execută în anumite momente în timp. Primul comportament de acest tip este *WakerBehaviour*. Această clasă abstractă implementează o sarcină de tip one-shot care se execută o singură dată, imediat după expirarea unui anumit timeout. *WakerBehaviour* are metodele `action()` și `done()` deja implementate pentru a executa metoda abstractă `onWake()` după expirarea unui anumit timeout (specificat în constructor). După executarea metodei `onWake()`, comportamentul se finalizează.

```
public class MyAgent extends Agent {
protected void setup() {
System.out.println("Adding waker behaviour");
addBehaviour(new WakerBehaviour(this, 10000) {
protected void onWake() {
// perform operation X
}
} );
}
}
```

În acest exemplu, operația X se efectuează la 10 secunde după ce textul dintre ghilimele este afișat.

8. Comportament de tip *TickerBehaviour*

Această clasă abstractă implementează o sarcină ciclică care se execută periodic, la intervale fixe de timp. TickerBehaviour are metodele `action()` și done() preimplementate pentru a executa metoda abstractă `onTick()` în mod repetitiv, așteptând o anumită perioadă (specificată în constructor) după fiecare execuție. Un comportament de tipul TickerBehaviour nu se termină niciodată decât dacă este eliminat sau dacă se apelează metoda `stop()`.

```
public class MyAgent extends Agent {
protected void setup() {
addBehaviour(new TickerBehaviour(this, 10000) {
protected void onTick() {
```

```
// perform operation Y
}
} );
}
}
```

Aici, operațiunea Y se efectuează periodic la fiecare 10 secunde.

9. Comportamente compozite

Această clasă abstractă implementează comportamente complexe care sunt alcătuite prin compunerea (secvențiere, execuție simulat paralelă sau înșiruire selectivă) unui număr de alte comportamente (copii). Astfel, sarcinile efectuate prin executarea acestui comportament nu sunt definite în comportamentul în sine, ci în interiorul comportamentelor acestuia (comportamente copil), în timp ce comportamentul compozit (părinte) are grijă doar de planificarea la execuție a copiilor conform unei anumite politici. De fiecare dată când metoda `action()` a unui comportament compozit este apelată, aceasta duce la apelarea metodei `action()` a unuia dintre copiii săi. Politica de programare determină ce copii să selecteze la fiecare execuție.

În special, clasa CompositeBehaviour oferă doar o interfață comună pentru planificarea execuției copiilor, dar nu definește nicio politică de programare. Această politică de planificare trebuie să fie definită de subclase (`SequentialBehaviour`, `ParallelBehaviour` și `FSMBehaviour`). O bună practică de programare este utilizarea numai a subclaselor clasei CompositeBehaviour, cu excepția cazului în care este necesară o politică specială de programare pentru copii (de exemplu, un comportament compozit în care planificarea se face pe bază de priorități, trebuie să extindă CompositeBehaviour direct).

a. Comportament secvențial

Această clasă implementează un comportament de tip compozit (`CompositeBehaviour`) care își execută subcomportamentele secvenţial și se termină când ultimul subcomportament este terminat. Este recomandată utilizarea acestui comportament atunci când o sarcină complexă poate fi exprimată ca o secvență de pași atomici (de exemplu, se efectuează un calcul/se ia o decizie, apoi se așteaptă primirea unui mesaj și se efectuează un alt calcul/se ia o altă decizie).

b. Comportament paralel

Clasa `ParallelBehaviour` implementează un comportament compus care își programează copiii în paralel și în această situație, programarea la execuție a comportamentelor copil este cooperativă și non-preemptivă. Aceasta înseamnă că de fiecare dată când este executată metoda `action()` a unui comportament paralel, aceasta invocă metoda `action()` a copilului curent și apoi mută pointerul curent înainte la următorul copil, indiferent dacă acesta din urmă a fost finalizat sau nu. Subcomportamentele într-un comportament paralel sunt adăugate prin metoda `addSubBehaviour()`. Un comportament paralel poate fi instruit să se termine atunci când toți copiii săi și-au terminat execuția sau, alternativ, când primul comportament copil își încheie execuția. Politica de terminare este selectată la momentul instanțierii prin specificarea în constructor, fie a constantei `WHEN_ALL` (pentru terminarea tuturor comportamentelor pentru definitivarea comportamentului paralel), fie a constantei `WHEN_ANY` (pentru terminarea a cel puțin unui comportament pentru definitivarea comportamentului paralel). Aceste constante definite în clasa `ParallelBehaviour`. O utilizare tipică a clasei `ParallelBehaviour` cu politica de terminare `WHEN_ANY` este de a anula o sarcină în cazul în care aceasta nu se finalizează într-un anumit interval, așa cum este exemplificat în codul următor:

```
Behaviour task = new MyTask();
ParellelBehaviour pb = new ParallelBehaviour(anAgent,
ParallelBehaviour.WHEN_ANY);
pb.addSubBehaviour(task);
pb.addSubBehaviour(new WakerBehaviour(anAgent, 60000) {
public void onWake() {
System.out.println("timeout expired");
}
});
```

c. Comportament de tip mașină cu stări

Această clasă implementează un comportament de tip compozit (CompositeBehaviour) care își execută copiii conform unei mașini cu stări finite, definită de utilizator (*Finite State Machine* – FSM). Mai detaliat, fiecare copil reprezintă activitatea care trebuie efectuată într-o stare a FSM, iar utilizatorul poate defini tranzițiile între stările FSM. Când copilul corespunzător stării S_i își termină sarcina pe care a avut-o de realizat, valoarea sa de terminare (așa cum este returnată de metoda `onEnd()`) este utilizată pentru a selecta tranziția către starea

următoare S_j. În runda de planificare a comportamentelor, copilul corespunzător stării S_j va fi executat. Unii dintre copiii unui comportament FSM pot fi înregistrați ca stări finale. Comportamentul FSM se încheie după finalizarea unuia dintre acești copii.

O detaliere suplimentară a modalității de lucru cu comportamentul de tip mașină cu stări finite se găsește în documentația javadoc a API-urilor JADE.

10. Executarea comportamentelor în fire de execuție dedicate

Planificarea execuției comportamentelor este efectuată într-un mod non-preemptiv, adică metoda `action()` a unui comportament nu este niciodată întreruptă pentru a permite altor comportamente să se execute. Numai când metoda `action()` a comportamentului curent își termină execuția, controlul este dat următorului comportament. Această abordare are mai multe avantaje în ceea ce privește performanța și scalabilitatea. Cu toate acestea, atunci când un comportament trebuie să efectueze unele operațiuni blocante (ex.: recepția unui mesaj pe interfața de rețea, comunicație pe socket), acesta blochează de fapt întregul agent și nu numai pe el însuși. O posibilă soluție la această problemă este, desigur, utilizarea firelor de execuție (*thread*) Java normale. Cu toate acestea, JADE oferă o soluție mai simplă prin intermediul comportamentelor de tip fir de execuție, adică acele comportamente care sunt executate în fire de execuție dedicate.

Orice comportament JADE (compozit sau simplu) poate fi executat într-un fir de execuție separat prin intermediul clasei `jade.core.behaviours.ThreadedBehaviourFactory`. Această clasă oferă metoda `wrap()`, care include un comportament JADE normal (*Behaviour*) într-un comportament de tip fir de execuție (`ThreadedBehaviour`). Adăugarea acelui comportament de tip `ThreadedBehaviour` la agent prin intermediul metodei `addBehaviour()` duce la executarea comportamentului original într-un fir de execuție dedicat. Trebuie observat că dezvoltatorii se ocupă doar de clasa `ThreadedBehaviourFactory`, în timp ce clasa `ThreadedBehaviour` este privată și nu este accesibilă.

Exemplul de cod următor arată cum se execută un comportament JADE într-un fir de execuție dedicat.

```
import jade.core.*;
import jade.core.behaviours.*;
public class ThreadedAgent extends Agent {
private ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();
protected void setup() {
// Crearea unui comportament JADE normal
Behaviour b = new OneShotBehaviour(this) {
public void action() {
// Perform some blocking operation that can take a long time
}
};
// Executarea comportamentului într-un fir de execuție separat
addBehaviour(tbf.wrap(b));
}
}
```

Comportamentele executate în cadrul firelor de execuție separate pot fi amestecate cu comportamente simple în interiorul comportamentelor compozite. De exemplu, un comportament secvențial de tip `SequentialBehaviour` poate avea doi copii executați sub formă de comportament simplu și un al treilea copil executat într-un fir de execuție dedicat. Comportamentul paralel (`ParallelBehaviour`) poate fi folosit pentru a atribui un grup de comportamente unui singur fir de execuție dedicat.

Există câteva elemente importante care trebuie luate în considerare atunci când se operează cu comportamente executate în fire de execuție dedicate:

- metoda `removeBehaviour()` a clasei Agent nu are niciun efect asupra comportamentelor din fire de execuție dedicate. Un comportament de acest tip este eliminat prin preluarea obiectului său de tip fir de execuție folosind metoda `getThread()` a clasei `ThreadedBehaviourFactory` și apelând metoda `interrupt()` a acestuia;
- când un agent este terminat, mutat sau își suspendă execuția, comportamentele de tip fir de execuție trebuie eliminate în mod explicit folosind tehnica descrisă anterior;
- atunci când un comportament de tip fir de execuție accesează unele resurse ale agentului părinte care sunt accesate și de alte comportamente cu sau fără fire de execuție dedicate, trebuie acordată atenția corespunzătoare problemelor de sincronizare/acces concurent la resurse.

Capitolul 5: Transmisia și recepția mesajelor în JADE

Cuprins

1. Metoda de comunicare

Standardizarea comunicației între agenți este o caracteristică fundamentală a JADE și este implementată în conformitate cu specificațiile FIPA (fipa.org).

Paradigma comunicației se bazează pe transmiterea asincronă a mesajelor. Astfel, fiecare agent are o coadă de mesaje/„cutie poștală“ în care platforma JADE postează mesaje recepționate de la alți agenți. Ori de câte ori un mesaj este postat în coada de mesaje, agentul destinatar este notificat. Cu toate acestea, când, sau dacă, agentul preia mesajul din coadă pentru procesare este o alegere de proiectare/implementare a dezvoltatorului. Acest proces este descris în figura 5.1.

Formatul special al mesajelor din JADE este conform cu cel definit de standardul FIPA-ACL (*Agent Communication Language* – http://www.fipa.org/repository/aclspecs.html). Fiecare mesaj include următoarele câmpuri:

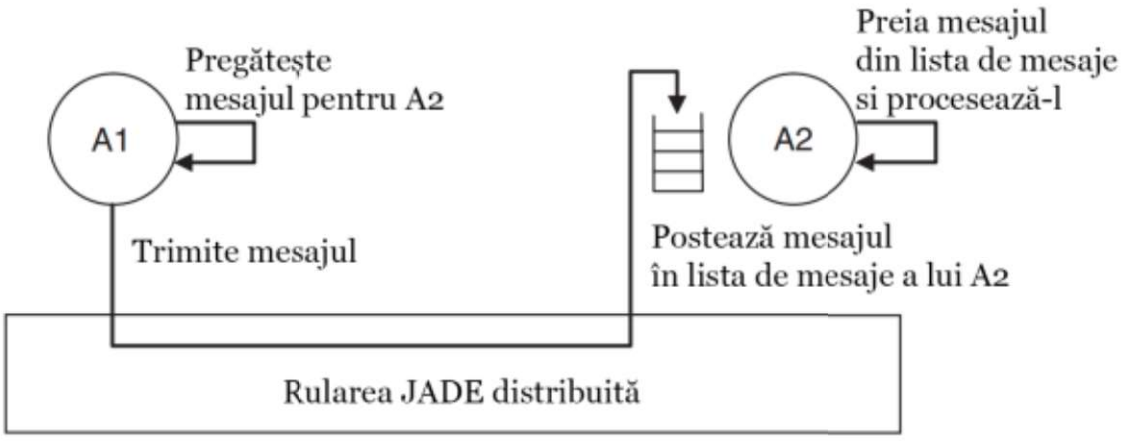

Fig.5.1 – Modalitatea de transmitere asincronă a mesajelor folosind platforma JADE

- expeditorul mesajului;
- lista receptorilor;
- actul comunicativ (*performative*) care indică ce intenționează să realizeze expeditorul prin trimiterea mesajului. Mai simplu, actul comunicativ poate fi asimilat tipului mesajului. De exemplu, dacă actul comunicativ este REQUEST (cerere), expeditorul dorește ca receptorul să efectueze o acțiune, dacă este INFORM, expeditorul dorește ca receptorul să fie informat asupra unui fapt, dacă este PROPOSE (propunere) sau CFP (*call for proposal*, apel la participare), expeditorul dorește să intre într-o negociere. Aceste tipuri sunt orientative și înțelesul final al mesajului este atribuit de dezvoltator în contextul aplicației realizate;
- corpul mesajului (*message content*) conține informațiile care urmează să fie schimbate prin mesaj (de exemplu, acțiunea care trebuie efectuată într-un mesaj de tip cererc; sau faptul că informația ce trebuie transmisă într-un mesaj INFORM etc.);
- limbajul de conținut (*content language*) care indică sintaxa utilizată pentru a exprima conținutul. Atât expeditorul, cât și receptorul trebuie să poată codifica și analiza expresii conforme cu această sintaxă pentru ca procesul de comunicare să fie coerent;
- ontologia care conține vocabularul simbolurilor utilizate în conținut. Atât expeditorul, cât și receptorul trebuie să atribuie aceleași semnificații acestor simboluri pentru a avea o comunicare coerentă;
- unele câmpuri suplimentare utilizate pentru a controla mai multe conversații simultane și pentru a specifica expirarea timpului pentru primirea unui răspuns, cum ar fi identificatorul (ID-ul) conversației, reply-with, in-reply-to și reply-by.

Un mesaj este implementat în JADE ca o instanță a clasei `jade.lang.acl.ACLMessage` care oferă metode de tip `get` și `set` pentru accesarea tuturor câmpurilor specificate de formatul ACL. Toate tipurile de mesaje (*performative*) definite în specificația FIPA sunt codificate ca niște constante în clasa `ACLMessage`.

2. Trimiterea mesajelor

Trimiterea unui mesaj către alt agent presupune completarea câmpurilor unui obiect `ACLMessage` și apoi apelarea metodei `send()` a clasei Agent. Codul următor creează un mesaj pentru a informa un agent al cărui identificator este Mihai că „Afară este înnorat“:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Mihai", AID.ISLOCALNAME));
msg.setLanguage("romana");
msg.setOntology("ontologie-previziune-vreme");
msg.setContent("Afara este innorat");
send(msg);
```

Tipurile de mesaje ACL definite de FIPA au o semantică formală bine definită, care poate fi exploatată pentru ca un agent să ia automat decizii adecvate atunci când este primit un mesaj. Pentru exemplificare, vom folosi tipuri de mesaje în interacțiunea multi-agent pe baza semnificației intuitive a lor. În continuare este prezentat un exemplu de sistem multi-agent în care se realizează tranzacții (achiziții) cu cărți. În particular, putem folosi tipul CFP (*call for proposal* – apel pentru propunere) pentru mesajele pe care agenții cumpărători le trimit agenților care oferă un serviciu pentru a solicita o ofertă. Tipul PROPOSE poate fi utilizat pentru mesajele care vin cu oferte pentru vânzător și tipul ACCEPT PROPOSAL pentru mesajele prin care se acceptă ofertele, adică ordinele de achiziție. În cele din urmă, tipul REFUSE va fi utilizat pentru mesajele trimise de agenții de tip vânzător atunci când serviciul/obiectul solicitat nu se află în catalogul de servicii/produse.

Pentru a menține lucrurile cât mai simple, vom pune titlul cărții de cumpărat în conținutul mesajelor CFP trimise de agenții cumpărător. În mod similar, conținutul mesajelor PROPOSE care transportă ofertele agenților de tip vânzător este prețul cărții. În cele ce urmează este dat un exemplu despre cum poate fi creat și trimis un mesaj de tip CFP de către un agent cumpărător:

```
// Mesaj cu conținutul unei oferte
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.lenght; ++i) {
// Adăugare agenți cărora să li se transmită mesajul
cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

3. Trimiterea mesajelor din interfața grafică

Pentru o testare simplă a unui agent cu comportamentul aferent de recepție a mesajelor, platforma JADE pune la dispoziție un set de unelte grafice pentru transmiterea de mesaje. Cele mai utilizate două unelte sunt interfața RMA și agenții de tip Dummy Agent (DA). Din RMA se poate trimite un mesaj unui agent prin selecția agentului respectiv, urmată de click dreapta și apoi Send Message (Fig.5.2). Alternativ, se poate folosi un agent de tip DA pentru a ajunge într-o fereastră similară de transmitere a mesajelor. În mod suplimentar, agentul DA permite și vizualizarea cozii de mesaje recepționate (Fig.5.3), putându-se astfel testa corectitudinea transmisiei conform diagramei din figura 5.4.

4. Primirea mesajelor

După cum s-a menționat anterior, platforma JADE transportă și depune automat mesaje în coada de mesaje private a agentului destinatar. Un agent poate prelua mesaje din coada sa de mesaje prin intermediul metodei `receive()`. Această metodă returnează primul mesaj din coada de mesaje (determinând astfel eliminarea acestuia) sau un obiect gol (null) în cazul în care coada de mesaje este goală. Având în vedere acest mod de funcționare se recomandă poziționarea metodei `receive()` într-un comportament de tip ciclic și procesarea locală a mesajelor. Trebuie avut totodată în vedere că recepția se face de agent (coadă mesaje) în timp ce procesarea se face într-un comportament. Acest lucru poate gernera probleme legate de vizibilitatea variabilelor ce conțin mesajul.

```
ACLMessage msg = receive();
if (msg != null) {
// Procesare mesaj
```

```
}
```

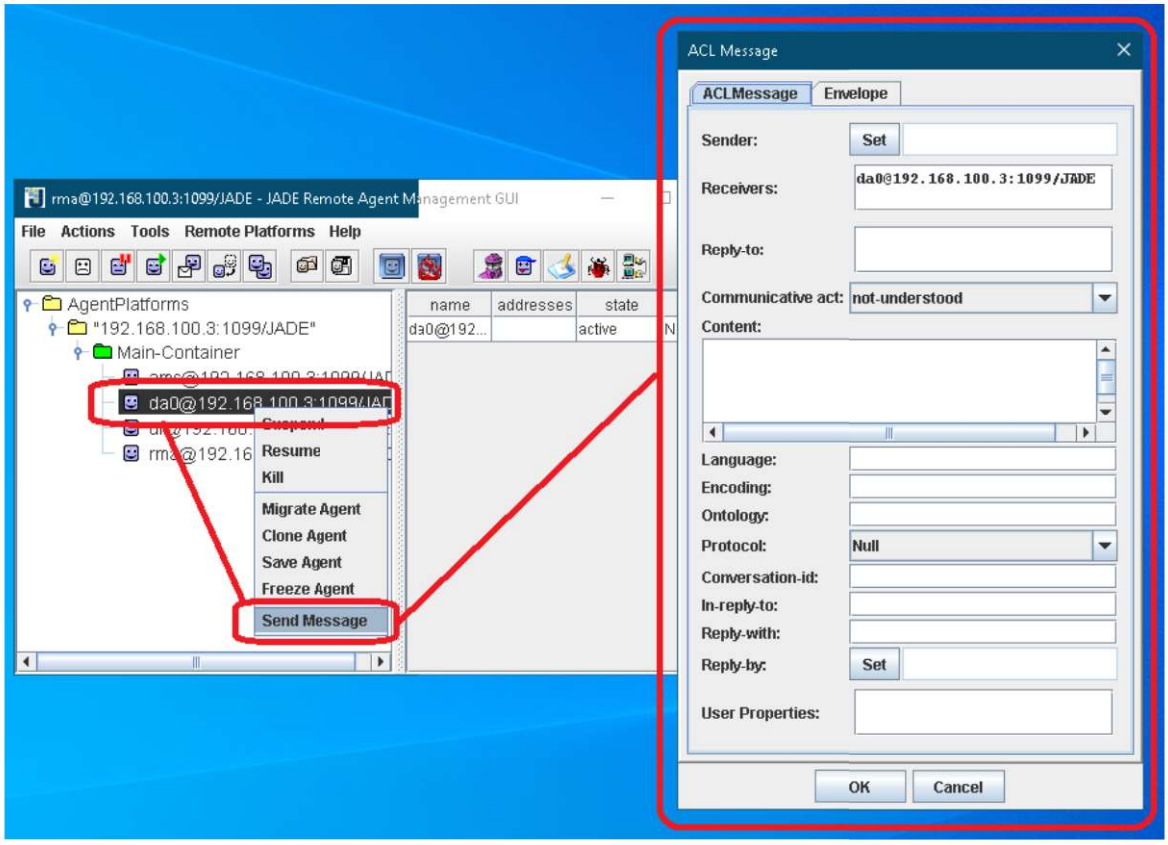

Fig.5.2 – Transmiterea unui mesaj din interfața RMA

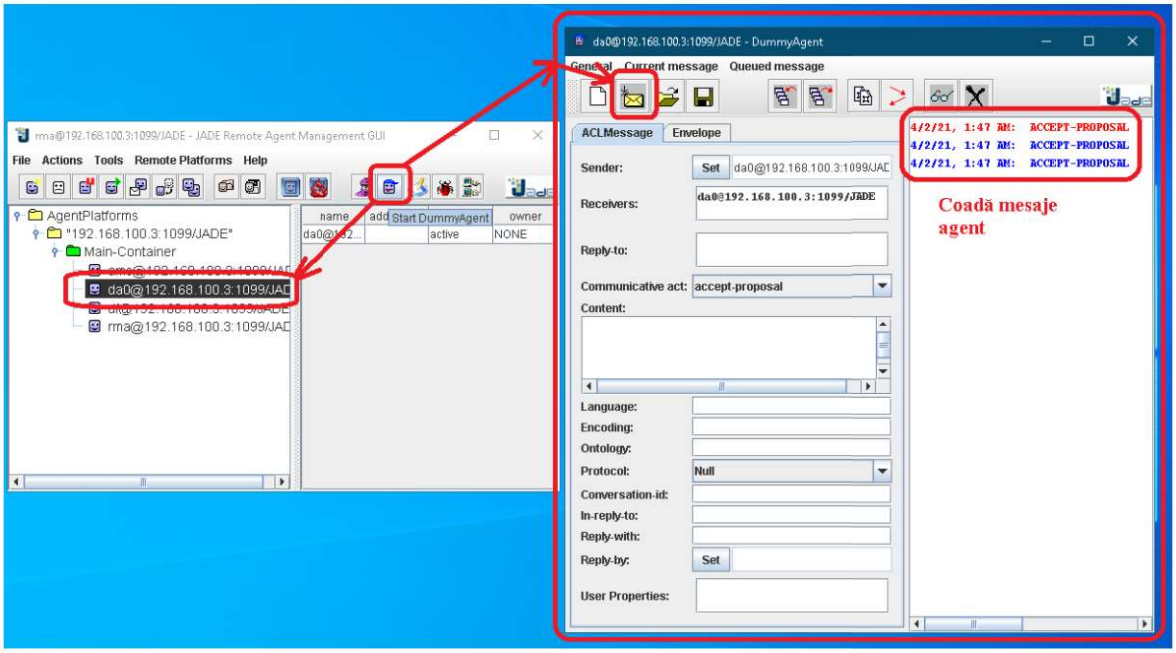

Fig.5.3 – Transmiterea unui mesaj folosind un agent de test de tip Dummy Agent

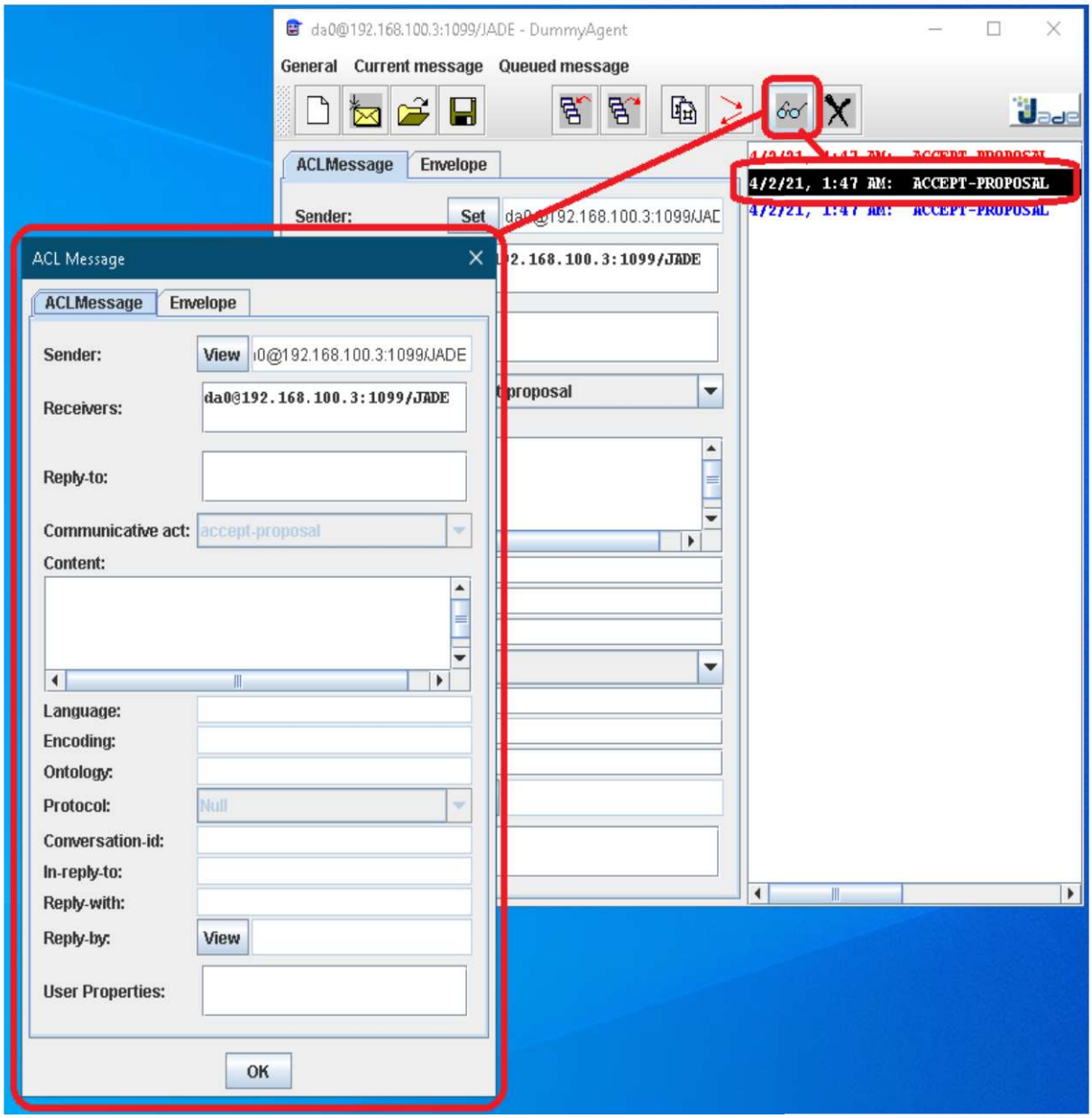

Fig.5.4 – Vizualizarea cozii de mesaje și a conținutului mesajelor

5. Testarea recepției mesajelor din interfața grafică

Cel mai simpu mod de a verifica expedierea corectă a unui mesaj este verificarea acestui proces folosind interfața grafică. În acest sens, se pot folosi uneltele/agenții Sniffer și Dummy Agent. În primul caz, agentul Sniffer captează toată comunicația cu observația că unul din capetele ei (expeditor sau destinatar) trebuie să fie monitorizat de agentul Sniffer. Monitorizarea se face dând click dreapta pe agentul dorit urmat de "Do sniff this agent" (Fig.5.5). Rezultatul acestei operații este că de fiecare dată când un mesaj vine sau pleacă de la agentul monitorizat, acesta va apărea ca o săgeată pe interfața grafică Sniffer. Dacă se face

dublu-click pe săgeata respectivă se va accesa conținutul mesajului. În acest sens, pentru monitorizarea protocoalelor de interacțiune complexe se recomandă crearea de configurații de rulare separate pentru participanți și inițiatori. Participanții trebuie lansați primii în execuție, urmați apoi de configurarea Sniffer pentru captarea mesajelor dorite și abia apoi de lansarea agenților de tip inițiator.

Fig.5.5 – Vizualizarea mesajelor sub forma unei diagrame secvențiale folosind interfața grafică a agentului Sniffer

Cea de-a doua modalitate de verificare a transmisiei corecte a unui mesaj este utilizarea unui agent de tip Dummy Agent pe post de receptor, urmată de inspecția cozii de mesaje a acestuia (Fig.5.4).

6. Reprogramarea comportamentului la execuție în vederea recepționării unui mesaj

Verificarea recursivă a cozii de mesaje în cadrul unui comportament ciclic generează un efort semnificativ pentru mașina pe care rulează agentul. Astfel, se recomandă reprogramarea selectivă a comportamentului pentru momentul în care un mesaj apare în coada de mesaje. Acest lucru se poate face folosind metoda `block`. Opțional se poate specifica și un timp maxim pentru reprogramare. Programatorii trebuie de obicei să implementeze comportamente care procesează mesajele primite de la alți agenți. Astfel de comportamente trebuie să ruleze continuu (comportamente ciclice) și, la fiecare execuție a metodei lor `action()`, trebuie să verifice dacă a fost primit un mesaj și să îl proceseze. În continuare este exemplificat comportamentul `CallForOfferServer`; sursa `PurchaseOrderServer` fiind disponibilă din biblioteca de exemple JADE.

```
/**
Implementara comportamentului CallForOfferServer ca o clasa internă
Acest comportament este folosit pentru recepționarea de mesaje de la
agenți de tip cumpărător
Dacă obiectul dorit este în repertoriu vânzătorul răspunde cu un mesaj
de tip PROPOSE în care se specifică prețul. Altfel, se răspunde cu un mesaj
de tip REFUSE.
*/
private class CallForOfferServer extends CyclicBehaviour {
public void action() {
ACLMessage msg = myAgent.receive();
if (msg != null) {
// Mesaj recepționat. Procesează
String title = msg.getContent();
ACLMessage reply = msg.createReply();
PriceManager pm = (PriceManager) catalogue.get(title);
if (pm != null) {
//Obiectul dorit este pe stoc. Prețul lui este pus în răspuns
reply.setPerformative(ACLMessage.PROPOSE);
reply.setContent(String.valueOf(pm.getCurrentPrice()));
}
else {
//Obiectul dorit nu este pe stoc.
reply.setPerformative(ACLMessage.REFUSE);
}
myAgent.send(reply);
}
}
} // Sfârșitul clasei interne
```

Ca de obicei, implementarea comportamentului `CallForOfferServer` este realizată sub forma unei clase interne a clasei `BookSellerAgent`. Această abordare simplifică lucrurile, deoarece putem accesa direct catalogul cărților de vânzare. Metoda `createReply()` a clasei ACLMessage creează automat un nou mesaj de tip ACL, setând automat receptorii și orice câmpuri necesare pentru controlul conversației (de exemplu, ID-ul conversației, reply-with și in-reply-by).

Cu referire la Figura 2.12, putem observa că, de îndată ce adăugăm comportamentul amintit, firul agentului începe o buclă infinită care folosește intensiv procesorul. Pe de altă parte, am dori ca metoda `action()` a comportamentului CallForOfferServer să fie executată numai atunci când este primit un mesaj nou. Pentru a efectua acest lucru, trebuie să folosim

metoda `block()` a clasei Behaviour, care, în ciuda a ceea ce sugerează numele metodei, nu este un apel de blocare, ci doar marchează comportamentul ca „blocat“, astfel încât agentul să nu îl mai programeze pentru executare. Când se introduce un mesaj nou în coada de mesaje a agentului, toate comportamentele blocate devin disponibile pentru executare din nou, astfel încât să aibă posibilitatea de a procesa mesajul primit. Prin urmare, metoda `action()` trebuie modificată după cum urmează:

```
public void action() {
ACLMessage msg = myAgent.receive();
if (msg != null) {
// Message received. Process it
...
}
else {
block();
}
}
```

Codul precedent este modelul tipic și recomandat pentru primirea mesajelor în interiorul unui comportament.

7. Primirea mesajelor în mod blocant

În afară de metoda `receive()`, clasa Agent oferă și metoda `blockingReceive()` care, așa cum sugerează și numele, este un apel blocant: nu returnează nimic până când nu există un mesaj în coada de mesaje a agentului. Este disponibilă și o versiune alternativă care primește un parametru de tip MessageTemplate (nu returnează nimic până când nu există un mesaj care să corespundă șablonului specificat).

Este important să subliniem că metoda `blockingReceive()` blochează de fapt firul de execuție al agentului. Prin urmare, dacă se apelează din interiorul unui comportament, acest lucru împiedică executarea tuturor celorlalte comportamente până când revine de la `BlockReceive()`. Luând în considerare acest lucru, o bună practică de programare este de a primi mesaje folosind `blockingReceive()` în metodele `setup()` și `takeDown()`, respectiv `receive()` în combinație cu `Behaviour.block()` în cadrul comportamentelor.

8. Citirea selectivă a mesajelor din coada de mesaje

În continuare este exemplificat modul de recepție a unui mesaj folosind un șablon de extracție din coada de mesaje:

```
            MessageTemplate        mt        =        MessageTemplate.and(
MessageTemplate.MatchPerformative(          ACLMessage.INFORM          ),
MessageTemplate.MatchSender( new AID( "a1", AID.ISLOCALNAME)));


            . . .


            ACLMessage msg = receive( mt );
            if (msg != null) { ... handle message }
            block();
```

9. Recepția unui mesaj folosind un timeout

În implementarea unui protocol de interacțiune se dorește ca în cazul unui defect (ex.: agent separat de platformă sau închis abrupt, din cauza unei defecțiuni tehnice a resursei pe care rulează), agentul care trebuie să recepționeze un mesaj să nu aștepte la infinit. Astfel, este nevoie de un comportament care să implementeze un mecanism de tip timeout: dacă un mesaj nu este primit într-un interval de timp, să întoarcă controlul execuției cu obiectul de tip mesaj (ACLMessage) vid. O implementare oarecum conformă cu specificațiile anterioare există în distribuția standard JADE, fiind vorba despre *ReceiverBehaviour*. În acest comportament se specifică un șablon și un timp de primire a mesajului (timeout). Comportamentul în cauză se termină fie când recepționează un mesaj, fie când expiră timpul de așteptare. Neajunsul comportamentului este acela că nu se poate specifica ce se întâmplă atunci când acesta se termină, fiind necesar să se păstreze o referință pentru a fi interogat ulterior și pentru a verifica metoda `done()` sau să se verifice mesajul primit (dacă este cazul). Această metodă de lucru poate genera interogări suplimentare. Alternativ, în cadrul unui protocol de interacțiune, se poate plasa comportamentul menționat într-un comportament secvențial și imediat după starea de recepție să urmeze o stare de verificare. În continuare este dat un exemplu de utilizare a comportamentului de tip ReceiverBehaviour în care sunt create două instanțe: una fără timeout (-1 la parametrul aferent) și una cu un timeout de 40 secunde.

```
private ReceiverBehaviour be1, be2;

    // Acest comportament așteaptă până la recepția unui mesaj
be1 = new ReceiverBehaviour(this, -1, null);
```

```
   addBehaviour(be1);

      // Acest comportamnet așteaptă 40 de secunde sau până la recepția
//unui mesaj de tip INFORM_REF

   be2 = new ReceiverBehaviour(this, 40000,
               MessageTemplate.MatchPerformative(ACLMessage.INFORM_REF));
   addBehaviour(be2);

//Ulterior, dacă comportamentul s-a terminat, mesajul poate fi extras într-
//o secțiune de cod try...catch
   ACLMessage msg;

   if (be1.done()) {
      try {
         msg = be1.getMessage();
         <... am primit mesaj la timp, îl analizăm...>
      }
      catch (ReceiverBehaviour.TimedOut e3) { ...Timed out!!! - nu am
primit mesaj...}
   }
   else
      < ...Nici timerul și nici mesajul nu au generat un eveniment...>
```

Ca urmare a inconvenientelor aferente comportamentului *ReceiverBehaviour*, în cele ce urmează vom prezenta implementarea unui comportament similar, dar care are posibilitatea de a specifica ce se rulează în momentul când apare unul din cele două evenimente de interes (recepție mesaj conform șablonului sau expirarea timpului de așteptare/timeout). Acest nou comportament include o metodă de procesare a mesajului apelată la terminare. Această metodă primește ca parametru mesajul sau NULL în cazul în care timeoutul expiră. Codul sursă și modalitatea de utilizare a clasei sunt următoarele:

```
//Cod sursă comportament de recepție cu timeout
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class myReceiver extends SimpleBehaviour
{

       private MessageTemplate template;
       private long timeOut, wakeupTime;
  private boolean finished;

  private ACLMessage msg;

  public ACLMessage getMessage() { return msg; }


  public myReceiver(Agent a, int millis, MessageTemplate mt) {
    super(a);
    timeOut = millis;
    template = mt;
  }
```

```
        public void onStart() {
                wakeupTime = (timeOut<0 ?
Long.MAX_VALUE:System.currentTimeMillis() + timeOut);
        }

        public boolean done () {
                return finished;
        }

        public void action()
        {
                if(template == null)
        msg = myAgent.receive();
                else
                        msg = myAgent.receive(template);

                if( msg != null) {
                        finished = true;
                        handle( msg );
                        return;
                }
       long dt = wakeupTime - System.currentTimeMillis();
       if ( dt > 0 )
         block(dt);
       else {
                        finished = true;
                        handle( msg );
       }
         }

        public void handle( ACLMessage m) { /* can be redefined in
sub_class */ }

        public void reset() {
                msg = null;
                finished = false;
                super.reset();
        }

        public void reset(int dt) {
                timeOut= dt;
                reset();
        }

}


//cod sursă pentru utilizarea comportamentului implementat mai sus
addBehaviour( new myReceiver(this, 40000,
         MessageTemplate.MatchPerformative(ACLMessage.INFORM_REF)
       {
          public void handle( ACLMessage msg )
          {
             if (msg == null)
                System.out.println("Timeout");
             else
                System.out.println("Mesaj primit: "+ msg);

          }
       });
```

10. Ștergerea mesajelor orfan

Pentru a preveni așteptarea infinită (în cadrul unui comportament de tip ciclic) este necesară utilizarea de timeouturi, însă pot apărea probleme cu mesajele care ajung târziu ca urmare a timpilor de procesare diferiți de 0 (procesarea unui mesaj nu se face instantaneu). Aceste mesaje, dacă nu sunt preluate din coadă la timp nu vor mai fi preluate deloc (ex.: filtrând după identificatorul comunicației, un mesaj neprocesat la timpul lui nu mai poate fi preluat de niciun comportament), devenind mesaje orfan. Acestea se acumulează în cozile de intrare ale agenților până când pot umple tot spațiul disponibil. De asemenea, încetinesc operațiunile – deoarece fiecare recepție cu un șablon trebuie să examineze mesajele orfane înainte de a ajunge la comportamentul aferent.

Timeouturile scurte de timp nu sunt singura cauză a mesajelor orfan. În urma unor teste, se poate constata că agenții primesc deseori mesaje de la platformă, mesaje care nu sunt rezultatul unei acțiuni personale, acestea ocupând spațiu și consumând timp de procesare. Astfel, este util să identificăm aceste mesaje și să le eliminăm din coada de mesaje. La procesul de debugging, este de asemenea util să afișăm mesajele orfan pentru a vedea dacă trebuie să prelungim timpii de procesare sau să adăugăm comportamente speciale pentru a le gestiona. Problema principală în procesarea mesajelor orfan este cum se diferențiază un mesaj orfan de unul care încă nu a fost procesat, dar urmează să fie. O soluție simplă la această problemă este să se considere mesaj orfan orice mesaj care a stat mai mult de un interval de timp în coada de mesaje. De obicei, acest interval de timp este ales în funcție de timpul de procesare maxim al agentului aferent (ex.: comportamentul cel mai lung al unui agent durează t_1, atunci timpul de verificare a mesajelor orfane este $t_2 >> t_1$). Pentru a face acest lucru, utilizăm un comportament de tip Ticker care ține evidența (cu un obiect de tip HashSet) tuturor mesajelor care erau în coadă ultima dată când a fost inspectată. Comportamentul citește (și elimină) toate mesajele din coadă. Dacă un mesaj a fost văzut la accesarea anterioară, este considerat mesaj orfan și nu va mai fi adăugat în coadă; în caz contrar, este plasat în setul de mesaje noi. Odată ce coada de mesaje este goală, putem returna mesajele procesate (și filtrate) în coadă folosind metoda `Agent.putBack(msg)`. Structura codului de citire și filtrare a mesajelor este ilustrată în figura 5.6.

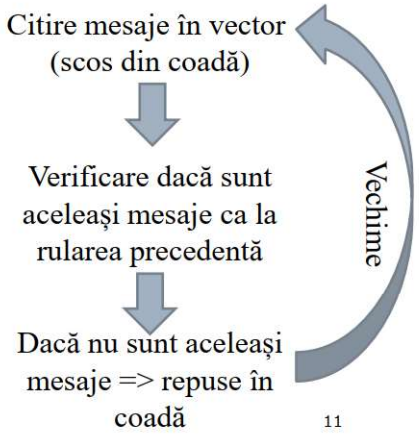

Fig.5.6 – Principiul de detectare și ștergere a mesajelor orfan
(care nu mai pot fi procesate de niciun comportament)

```
class GCAgent extends TickerBehaviour
   {
      Set seen = new HashSet(),
          old  = new HashSet();

      GCAgent( Agent a, long dt) { super(a,dt); }

      protected void onTick()
      {
         ACLMessage msg = myAgent.receive();
         while (msg != null) {
            if (! old.contains(msg))
               seen.add( msg);
            else {
               System.out.print("+++ Flushing message: ");
               dumpMessage( msg );
            }
            msg = myAgent.receive();
         }
         for( Iterator it = seen.iterator(); it.hasNext(); )
            myAgent.putBack( (ACLMessage) it.next() );

         old.clear();
         Set tmp = old;
         old = seen;
         seen = tmp;
      }
   }
```

11. Metode de găsire a agenților cu care se interacționează

Un lucru absolut necesar în trimiterea unui mesaj îl reprezintă aflarea numelui local al agentului receptor. Folosind numele local, se poate forma identificatorul agent folosit în câmpul receptor. În acest sens există patru posibilități:

- numele local este cunoscut la rulare: ca urmare a lansării în execuție din linia de comandă, se știe *a priori* numele instanței agentului cu care dorim să dialogăm. Acest nume poate fi trimis ca parametru agentului emițător;
- nume local cunoscut ca urmare a faptului că se răspunde la un mesaj primit: în cazul în care mesajul ACL este asamblat folosind metoda `createReply()`, destinatarul și emițătorul sunt schimbați în mod automat;
- căutare pe platformă după nume sau după o parte din nume: agentul AMS (responsabil cu gestiunea platformei multi-agent) poate fi interogat pentru a obține un vector cu numele agenților existenți pe platformă. Această metodă poate fi folosită atunci când se dorește să se transmită un mesaj tuturor agenților, dar trebuie avut grijă să nu se transmită acest mesaj și agentului curent care se află în lista de agenți descoperiți pe platformă. În continuare este dată o secțiune de cod care prezintă modalitatea de interogare a platformei.

```
import jade.domain.AMSService;
import jade.domain.FIPAAgentManagement.*;
    ...
AMSAgentDescription [] agents = null;
try {
    SearchConstraints c = new SearchConstraints();
    c.setMaxResults ( new Long(-1) );
 agents = AMSService.search( this, new AMSAgentDescription (), c );
}
catch (Exception e) { ... }

AID myID = getAID();
for (int i=0; i<agents.length;i++)
{
    AID agentID = agents[i].getName();
    System.out.println(
        ( agentID.equals( myID ) ? "*** " : "    ")
        + i + ": " + agentID.getName()
    );
}
```

- căutare pe *Pagini aurii* (DF – *Directory Facilitator*), după serviciul software implementat de agent (Fig.5.7).

Un serviciu de tip *Pagini aurii* permite agenților să publice descrieri ale unuia sau mai multor servicii pe care le oferă pentru ca alți agenți să le poată descoperi și exploata cu ușurință. O interacțiune standardizată între doi agenți se numește serviciu. Agentul participant, cel care oferă serviciul, se poate înregistra pe platforma DF, el putând fi ulterior căutat de către agenți care doresc să interacționeze cu el. În acest sens, se poate folosi o structură de date de tip `DFAgentDescription` atât pentru căutarea cât și pentru înregistrarea agenților.

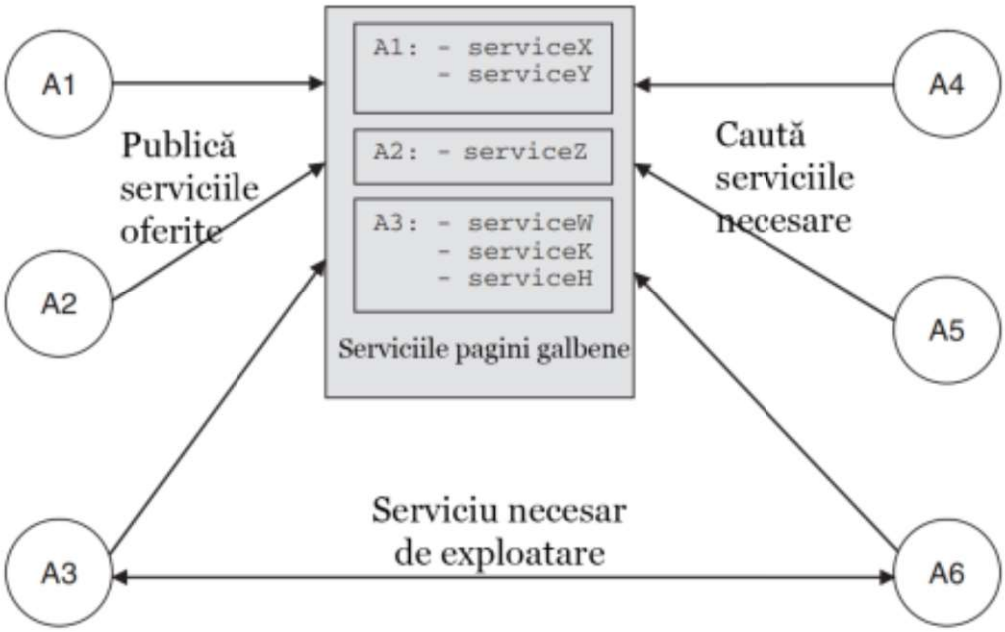

Fig.5.7 – Modalitatea de înregistrare, căutare și accesare a serviciilor folosind DF

Serviciul de *Pagini aurii* din JADE, în conformitate cu specificațiile FIPA Agent Management, este furnizat de un agent specializat numit DF (*Directory Facilitator*). Fiecare platformă compatibilă cu FIPA trebuie să găzduiască un agent implicit DF (al cărui nume local este 'df@nume-platformă'). Alți agenți DF pot fi implementați dacă este necesar și mai mulți agenți DF (inclusiv cel implicit) pot fi grupați pentru a oferi un serviciu tolerant la defect.

Deoarece DF este un agent, este posibil să se interacționeze cu el ca și cu orice alt agent prin schimbul de mesaje ACL, utilizând un limbaj de conținut și o ontologie adecvată (de exemplu, ontologia FIPA-agent-management ontology) așa cum sunt definite prin specificațiile FIPA. Pentru a simplifica aceste interacțiuni, JADE oferă clasa `jade.domain.DFService` cu care este posibil să se publice și să se caute servicii printr-o varietate de metode.

În cele ce urmează sunt ilustrate exemple pentru procesele de înregistrare, ștergere și căutare pe DF.

a) Publicare/înregistrare servicii agent

Un agent care dorește să publice servicii trebuie să furnizeze DF-ului o descriere care include propriul identificator agent (AID), tipul, numele, gramatica și ontologia pe care alți agenți trebuie să le folosească pentru a interacționa cu acesta. Clasele `DFAgentDescription`, `ServiceDescription` și `Property`, incluse în pachetul `jade.domain.FIPAAgentManagement`, reprezintă aceste abstractizări.

Pentru a publica un serviciu, un agent trebuie să creeze o descriere adecvată (ca instanță a clasei `DFAgentDescription`) și să apeleze metoda statică `register()` a clasei `DFService`. Pentru cazul unui sistem multi-agent în care se tranzacționează produse, agenții cumpărător își înregistrează capacitatea de a cumpăra produse (un serviciu de tip „cumpărare produse“) în metoda `setup()`, după cum urmează:

```
//Înregistrare agent pe DF cu serviciul cumpărător
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName( getAID() );
        ServiceDescription sd  = new ServiceDescription();
        sd.setType( "cumpărător" );
        sd.setName( getLocalName() );
        dfd.addServices(sd);

        try {
            DFService.register(this, dfd );
        }
        catch (FIPAException fe) { fe.printStackTrace(); }


//Stergere agent de pe DF - este recomandat ca odată cu închiderea
//agentului (doDelete) să se șteargă și înregistrarea pe DF
protected void takeDown()
       {
          try { DFService.deregister(this); }
          catch (Exception e) {}
       }
```

b) Căutare servicii agent

Un agent care dorește să caute servicii trebuie să furnizeze DF-ului o descriere a șablonului de căutare. Rezultatul căutării este o listă cu toate descrierile care se potrivesc cu șablonul furnizat. Conform specificațiilor FIPA, o descriere se potrivește cu șablonul dacă toate câmpurile specificate în șablon sunt prezente în descriere cu aceleași valori. Metoda statică de căutare `search()` a clasei `DFService` poate fi utilizată așa cum este exemplificat în continuare:

```
//Căutare agent pe DF
            DFAgentDescription dfd = new DFAgentDescription();
            ServiceDescription sd  = new ServiceDescription();
            sd.setType( "buyer" );
            dfd.addServices(sd);

            DFAgentDescription[] result = DFService.search(this, dfd);

            System.out.println(result.length + " results" );
            if (result.length>0)
                System.out.println(" " + result[0].getName() );
```

Legat de căutarea pe DF, se poate face o interogare fără a specifica numele unui serviciu, ci doar tipul acestuia. În această situație, este returnat un vector de dimensiune 0 dacă nu există înregistrări care să corespundă.

c) Excluderea unui agent din *Pagini aurii* (dezabonare – *deregistering*)

Când un agent este terminat, este o bună practică să se șteargă intrarea sa din DF. Deși aceste intrări sunt eliminate automat din AMS atunci când un agent este terminat, sistemul nu le elimină automat din DF, această sarcină rămânând în sarcina dezvoltatorului să o facă în mod explicit. Acest lucru este important deoarece fiecărui agent i se permite o singură intrare în DF, iar încercările de înregistrare multiplă a unui agent în DF generează o eroare/excepție.

Modul obișnuit de a exclude un agent cu serviciile aferente din DF este în metoda `takeDown()`, apelată în mod automat la terminarea normală a agentului. În cele ce urmează este dat un exemplu de excludere din DF:

```
protected void takeDown()
{
   try { DFService.deregister(this); }
   catch (Exception e) {}
}
```

Chiar și cu această precauție, este o problemă obișnuită ca atunci când se repornește un agent care a fost terminat anterior, înregistrarea acestuia să eșueze deoarece vechea înregistrare a acestuia în DF este încă în actuală. În acest sens, se recomandă verificarea înregistrărilor vechi și eliminarea acestora înainte de adăugarea înregistrării noi. În continuare este dată o secțiune de cod care automatizează acest proces:

```
void register( ServiceDescription sd)
{
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());

try {
        DFAgentDescription list[] = DFService.search( this, dfd );
                if ( list.length>0 )
                        DFService.deregister(this);

        dfd.addServices(sd);
        DFService.register(this,dfd);
}
catch (FIPAException fe) { fe.printStackTrace(); }
}
```

12. Interacțiuni simple

În figura 5.8 este dat un exemplu de interacțiune simplă într-un SMA. În acest exemplu este nevoie de minimum doi agenți: un agent participant și un agent inițiator. Participantul răspunde la un mesaj primit de tip PROPOSE (conform standardului FIPA-ACL), în timp ce inițiatorul are două sarcini: prima dată trebuie să își adauge comportamentul de recepție a unui mesaj de tip ACCEPT/ REJECT-PROPOSAL în așteptarea răspunsului de la participant și apoi să identifice participantul (v. Secțiunea 11 – Metode de găsire a agenților cu care se interacționează) și să inițieze interacțiunea printr-un mesaj de tip PROPOSE. Pentru a putea garanta această funcționare este recomandat să se realizeze două configurații de rulare: prima configurație pornește platforma împreună cu participantul care trebuie să fie gata în momentul sosirii mesajului (adițional, se poate porni agentul *Sniffer* pentru monitorizarea comunicației); cea de-a doua configurație atașează un container nou la platformă și lansează agentul inițiator. Acest mod de rulare garantează că agentul inițiator are cui să îi trimită un mesaj.

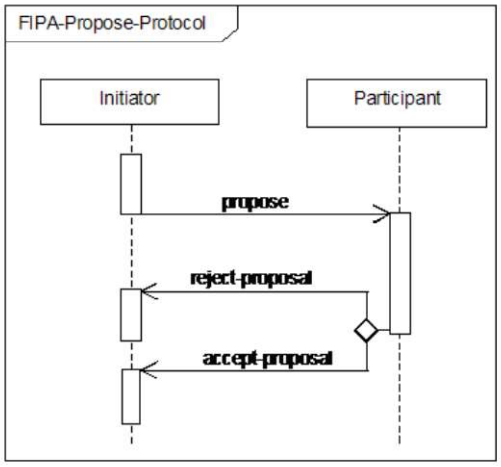

Fig.5.8 – Interacțiune simplă între doi agenți (conform protocolului de interacțiune FIPA)

În situația în care se încearcă trimiterea unui mesaj către un agent care nu există, platforma (prin agentul AMS) va informa inițiatorul printr-un mesaj de tip FAILURE că trimiterea a eșuat. Iată ilustrată o secțiune din conținutul mesajului FAILURE:

```
(internal-error "Agent not found: getContainerID() failed to find agent
nimic@192.168.100.5:1099/JADE")))
```

Capitolul 6: Conversații complexe și protocoale de interacțiune

Cuprins

1. Introducere

Unul dintre principalele aspecte în dezvoltarea sistemelor multi-agent este reprezentat de comunicarea între agenți. JADE oferă acest lucru ca una dintre caracteristicile sale fundamentale, iar implementarea sa este în conformitate cu standardele FIPA. Comunicarea dintre agenții JADE se bazează pe un schimb asincron de mesaje, fiecare agent având o coadă internă în care sunt stocate mesajele trimise de alți agenți. Chiar dacă agentul este notificat când un mesaj a fost adăugat în coadă, este alegerea dezvoltatorului când sau dacă mesajele sunt preluate și ulterior procesate.

Structura mesajelor este conformă standardului FIPA-ACL și conține următoarele elemente: expeditorul mesajului, lista destinatarilor, actul comunicativ (tipul mesajului), conținutul mesajului, limbajul conținutului, ontologia, precum și câteva câmpuri suplimentare care permit controlul conversațiilor simultane, cum ar fi identificatorul conversației (*conversationid*), o expresie care trebuie utilizată de un agent care răspunde pentru a identifica mesajul (*reply-with*), o referință la o acțiune anterioară la care mesajul este un

răspuns (*in-reply-to*), o dată care indică până când trebuie primit un răspuns (*reply-by*). Limbajul conținutului se referă la sintaxa utilizată pentru exprimarea conținutului mesajului, în timp ce ontologia se referă la semnificațiile asociate acestui conținut.

2. Interacțiuni complexe și utilizarea comportamentelor compozite

În JADE, sarcinile agent sunt implementate în metodele `action()` și `done()` ale instanțelor clasei `jade.core.behaviours.Behaviour` sau a unor clase derivate. Când un agent realizează sarcini complexe care implică mai mulți pași de calcul, posibil intercalați cu conversații cu alți agenți sau alte tipuri de interacțiuni paralele, nu este convenabil să se implementeze logica sarcinii de realizat în cadrul metodei `action()` a unui singur comportament, aceasta devenind excesiv de mare și dificil de gestionat.

O abordare mai simplă și mai ușor de gestionat pentru a implementa sarcini complexe în JADE este utilizarea de comportamente compozite. Astfel, sarcini simple sunt înseriate sau executate în paralel ca parte a unor comportamente de tip secvențial, mașină de stări sau paralel. Rolul comportamentelor compozite este de a compune conportamentele simple conform unei logici determinate de dezvoltator. Baza pentru această caracteristică este oferită de clasa `CompositeBehaviour` inclusă în pachetul `jade.core.behaviours`. Un comportament compozit (o instanță a clasei `CompositeBehaviour`) este el însuși un comportament care încorporează un număr de subcomportamente copil. Clasa `CompositeBehaviour` implementează deja metoda `action()` în așa fel încât, de fiecare dată când este apelată, invocă metoda `action()` a unuia dintre copiii săi. Politica utilizată pentru a selecta copilul, care se declanșează la fiecare rundă, este delegată metodelor: `scheduleFirst()`, responsabilă cu prima rundă de execuție și `scheduleNext()` responsabilă cu rundele succesive. Aceste metode sunt declarate de tip abstract și sunt definite în subclasele `CompositeBehaviour` care implementează tipurile reale de comportamente compozite.

Trei tipuri de comportamente compozite gata de utilizat sunt furnizate în distribuția standard JADE (Fig.6.1). Acestea sunt: `SequentialBehaviour` – pentru realizarea de scheme de execuție secvențiale, `FSMBehaviour` – pentru realizarea de scheme de execuție de tip mașină de stare (similar execuției secvențiale, dar cu posibilitatea de a realiza salturi

între oricare două stări) și `ParallelBehaviour` – pentru realizarea de scheme de execuție paralele. Aceste trei tipuri de comportamente, precum și modalitatea de a schimba informații între comportament ele vor fi detaliate în următoarele secțiuni. Subcomportamentele încorporate într-un comportament compozit pot fi, de asemenea, comportamente compozite, făcând astfel posibilă crearea unor sarcini complexe prin combinarea ierarhică a comportamentelor simple. În mod uzual, dezvoltatorii nu au nevoie să extindă direct `CompositeBehaviour`, ci pot folosi doar subclasele sale `SequentialBehaviour`, `FSMBahaviour` și `ParallelBehaviour`.

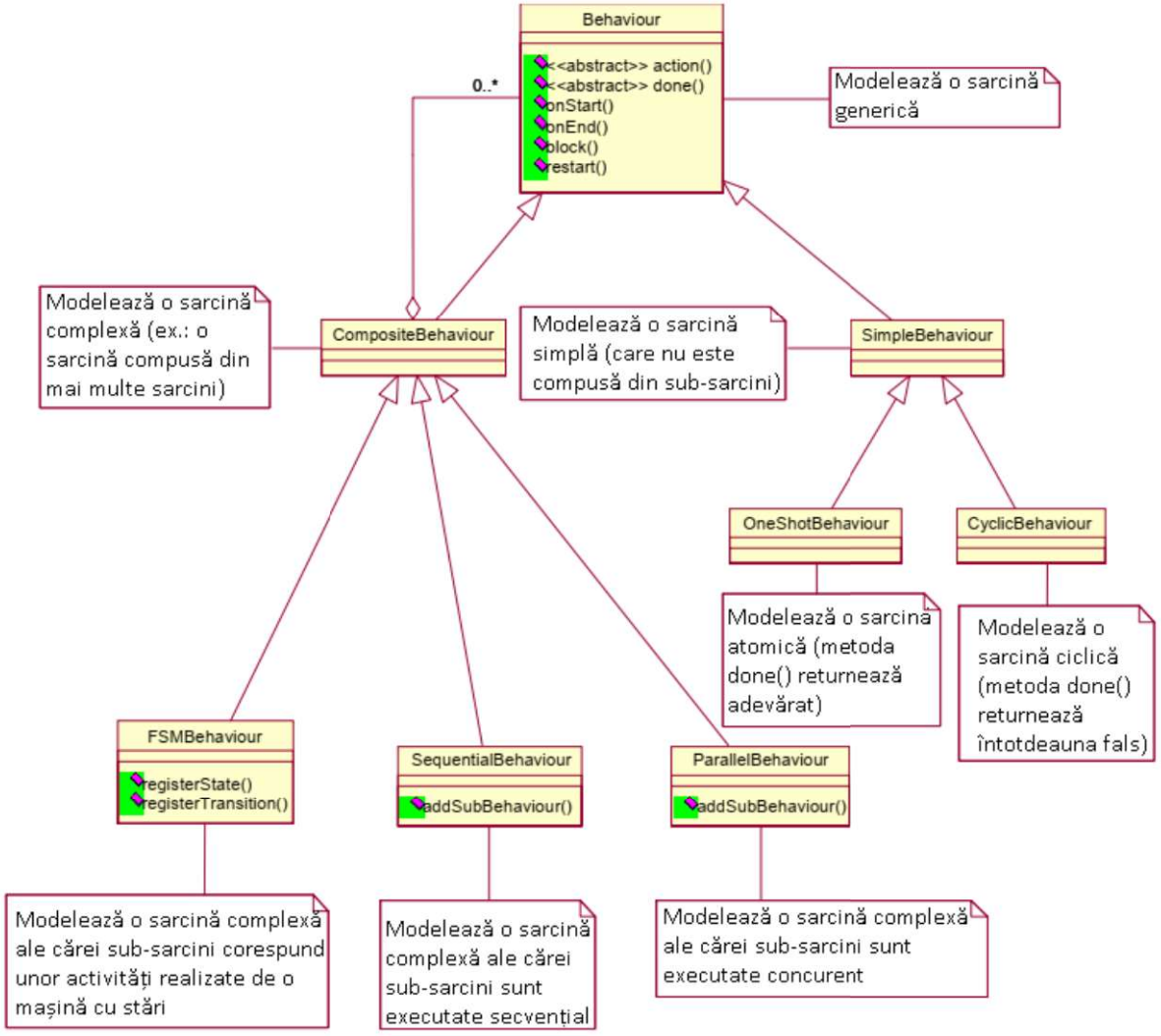

Fig.6.1 – Ierarhia clasei `Behaviour` (comportament)

3. Comportament secvențial

Clasa `SequentialBehaviour` implementează un comportament compozit care își programează comportamentele copil în conformitate cu o politică secvențială simplă. Începe

cu primul comportament copil; când acesta se termină (adică metoda `done()` returnează adevărat), se trece la următorul comportament copil și așa mai departe. Când ultimul comportament copil este finalizat, întregul comportament secvențial se încheie. În figura 6.2 este prezentat fluxul de operații executate de fiecare dată când comportamentul secvențial este planificat și metoda sa `action()` este executată.

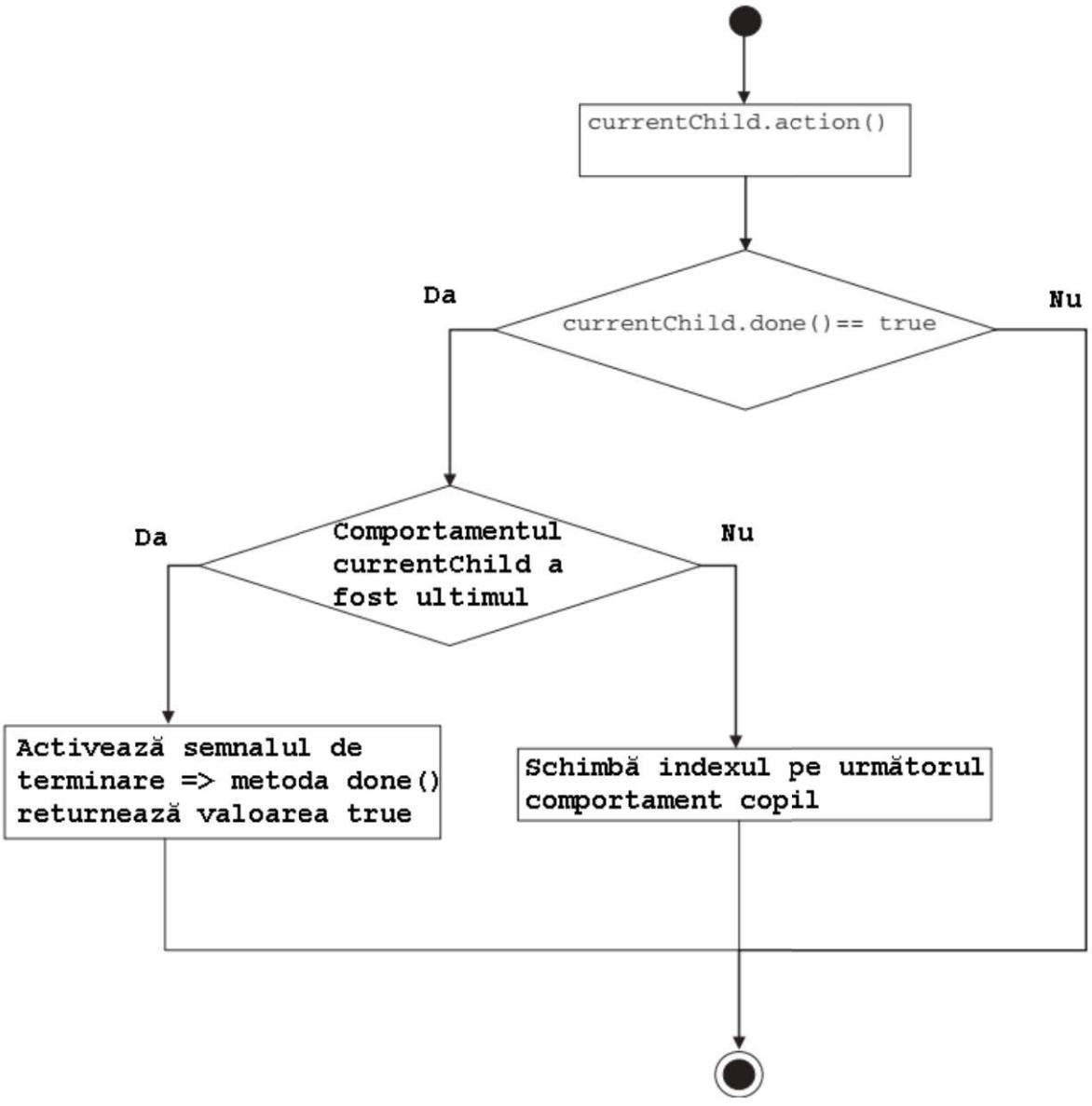

Fig.6.2 – Fluxul de instrucțiuni pentru comportamentul `SequentialBehaviour`

Subcomportamentele (comportamentele copil) într-un comportament secvențial sunt adăugate folosind metoda `addSubBehaviour()`. Ordinea în care sunt adăugate sub-comportamentele determină ordinea în care sunt programate de comportamentul secvențial. De exemplu, un comportament cu trei stări succesive poate fi implementat în mod convenabil folosind clasa `SequentialBehaviour`, după cum urmează:

```
      SequentialBehaviour threeStepBehaviour = new
SequentialBehaviour(anAgent);
      threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
      public void action() {
      // realizarea operației X
      }
      } );
      threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
      public void action() {
      // realizarea operației Y
      }
      } );
      threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
      public void action() {
      // realizarea operației Z
      }
      } );
```

Execuțarea comportamentului menționat asigură execuțarea celor trei operații (X, Y, Z) în ordinea în care au fost adăugate.

4. Comportament de tip mașină cu număr finit de stări (FSM)

Clasa `FSMBehaviour` implementează un comportament compozit care își programează comportamentele copil/ subcomportamentele în conformitate cu o mașină cu un număr finit de stări (*Finite State Machine* – FSM) care corespund cu comportamentele copil ale comportamentului FSM. Clasa `FSMBehaviour` oferă metode pentru a înregistra sub-comportamente ca stări ale mașinii finite și pentru a înregistra tranziții între aceste stări. Similar unui comportament secvențial, un comportament FSM păstrează un pointer către comportamentul copil actual. Când comportamentul copil curent își termină execuția (metoda `done()` returnează valoarea adevărat), comportamentul FSM își verifică tabelul de tranziție intern și, pe baza acestuia, selectează noul comportament copil ce va fi programat la execuție data viitoare când se execută metoda `action()`. Tranzițiile într-un comportament FSM sunt identificate prin numere întregi, alese ca urmare a execuției metodei `onEnd()`. Când comportamentul copil curent al comportamentului FSM este finalizat, valoarea returnată de metoda `onEnd()` este luată ca valoare de ieșire și, pe baza ei, se verifică toate tranzițiile care ies din starea/comportamentul copil curent. Prima tranziție a cărei etichetă se potrivește cu

valoarea returnată de metoda `onEnd()` este activată și starea destinație a acesteia devine noul comportament copil actual. Metoda `registerState()`, utilizată pentru a adăuga stări la o instanță `FSMBehaviour`, acceptă două argumente: un șir de caractere care definesc numele stării înregistrate și un comportament care va fi executat în acea stare. Metoda `registerTransition()`, utilizată pentru a adăuga tranziții la o instanță `FSMBehaviour`, acceptă trei argumente: două șiruri de caractere care definesc starea sursă și starea destinație a tranziției și o valoare de tip intreg ce definește eticheta care marchează tranziția. Metodele `registerFirstState()` și `registerLastState()` realizează înregistrarea stării inițiale (de intrare), respectiv înregistrarea stărilor finale ale FSM. Un comportament de tip FSM are o singură stare de intrare, dar poate avea mai multe stări finale. Întregul comportament FSM se încheie când se ajunge la o stare finală și se execută complet. În scop didactic, în cele ce urmează, este ilustrat modul de realizare al unui comportament FSM care respectă modul de operare din figura 6.3.

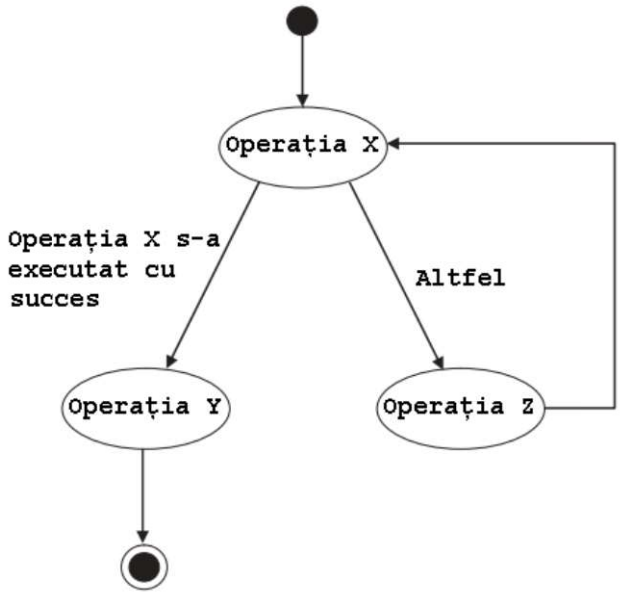

Fig.6.3 – Exemplu de mașină cu un număr finit de stări (comportament compozit de tip FSM)

```
FSMBehaviour sampleFSM = new FSMBehaviour(anAgent);
sampleFSM.registerFirstState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation X
}
public int onEnd() {
return (operation X successful ? 1 : 0);
}
}, "X");
```

```
sampleFSM.registerLastState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation Y}
}, "Y");
sampleFSM.registerState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation Z
}
}, "Z");
sampleFSM.registerTransition("X", "Y", 1);
sampleFSM.registerTransition("X", "Z", 0);
sampleFSM.registerDefaultTransition("Z", "X", new String[]{"X",
"Z"});
```

Metoda `registerDeafultTransition()` a clasei `FSMBehaviour` permite definirea unei tranziții implicite între două stări. O tranziție implicită nu este marcată cu nicio etichetă și este realizată dacă și numai dacă nu sunt active alte tranziții (dacă există) care ies din starea curentă. Atât metoda `registerTransition()`, cât și metoda `registerDefaultTransition()` au implementări alternative în care pot primi ca parametru de tip șir de caractere. Acest parametru indică un set de comportamente copil ale comportamentului FSM care trebuie resetate atunci când este activată tranziția înregistrată. Acest lucru este util atunci când se înregistrează „tranziții înapoi", adică tranziții care conduc la stări care au fost deja vizitate. Mai mult, platforma JADE operează astfel încât orice obiect de tip `Behaviour` (comportament) care a fost executat o dată trebuie resetat apelând metoda `reset()` înainte de a putea fi executat din nou. De exemplu, cu referire la figura 6.3, dacă se urmează tranziția de la Z la X, starea X și eventual starea Z vor fi executate încă o dată. În urma execuției, acestea trebuie resetate pentru a evita efecte nedorite.

5. Comportament paralel

Clasa ParallelBehaviour implementează un comportament compozit care își programează comportamentele copil/subcomportamentele în paralel. Ca de obicei, atunci când avem de-a face cu comportamente JADE, programarea este cooperativă și nu preemptivă, iar termenul paralel reflectă modalitatea de terminare a comportamentului (când toate subcomportamentele s-au terminat) și nu execuția în paralel a subcomportamentelor, acest lucru nefiind posibil deoarece fiecare agent este asociat în mod implicit unui singur fir de execuție. Aceasta înseamnă că de fiecare dată când se execută metoda `action()` a unui

comportament paralel, se invocă metoda `action()` a comportamentului copil curent și apoi mută indicatorul înainte către următorul comportament copil, indiferent dacă acesta din urmă a fost finalizat sau nu. Subcomportamentele într-un comportament paralel sunt adăugate prin intermediul metodei `addSubBehaviour()`. Adăugarea de subcomportamente într-un comportament paralel se face folosind metoda `addSubBehaviour()`. Politica de terminare este selectată la momentul instanțierii prin specificarea în constructor a constantelor `WHEN_ALL` sau `WHEN_ANY` definite în clasa `ParallelBehaviour`. O utilizare tipică a clasei `ParallelBehaviour` cu politica de terminare `WHEN_ANY` este de a anula o sarcină în cazul în care aceasta nu se finalizează într-un anumit interval de timp, după cum se exemplifică în codul următor:

```
Behaviour task = new MyTask();
ParellelBehaviour pb = new ParallelBehaviour(anAgent,
ParallelBehaviour.WHEN_ANY);
pb.addSubBehaviour(task);
pb.addSubBehaviour(new WakerBehaviour(anAgent, 60000) {
public void onWake() {
System.out.println("timeout expired");
}
});
```

Politica `WHEN_ANY` se folosește în situația în care terminarea comportamentului paralel necesită terminarea tuturor comportamentelor copil. Un exemplu de astfel de situație este așteptarea unui set de mesaje (sau expirarea timeout-urilor aferente) de la mai mulți participanți la conversație. Această situație particulară va fi detaliată în cadrul exemplului de la sfârșitul acestui capitol.

6. Utilizarea de comportamente compozite pentru implementarea de interacțiuni complexe

Un model de scenariu des utilizat în sistemele multi-agent este acela în care un agent trimite un mesaj și așteaptă un răspuns. Trimiterea și recepția de mesaje este un lucru simplu, dar situația se complică atunci când trebuie să așteptăm, nu orice mesaj, ci un mesaj de la agentul căruia i-am trimis cererea și, în plus, să fie un răspuns la cererea noastră inițială. Acesta este un model comun de interacțiune și FIPA a definit un set extins de protocoale de interacțiune pentru aceste situații clasice. JADE oferă clase generice care implementează protocoale FIPA, precum: AchieveREInitiator, AchieveREResponder, SimpleAchieveREInitiator, SimpleAchieveREResponder, ContractNetInitiator, ContractNetResponder

Problema cu aceste clase este că sunt prea complexe de înțeles și dificil de utilizat. Pentru sistemele industriale cu arhitecturi deschise în care trebuie să interacționeze agenți creați de diferite organizații, aceste protocoale generice FIPA sunt soluția; dar pentru proiecte de dimensiuni reduse este mai ușor și mai clar să se implementeze interacțiunile necesare folosind comportamente simple și compozite. În cele ce urmează vor fi acoperite ambele situații: a) implementare protocol interacțiune – secțiune 7, respectiv b) folosirea unui protocol generic de interacțiune – secțiune 10.

JADE oferă un set de comportamente utile care pot fi extinse pentru a modela activitatea complexă tipică agenților reali. Practic, există două tipuri de clase comportament: de bază/primitive, în această clasă intrând comportamentele simple sau ciclice, și compozite /compuse din comportamente, care pot combina atât comportamente simple, cât și compozite pentru a se executa secvențial sau în paralel.

În cadrul acestei secțiuni se va detalia modalitatea de implementare a unei interacțiuni 1 (inițiator) – m (participanți) folosind comportamente de tip paralel și secvențial. Se va considera o situație aferentă domeniului E-commerce, în care un agent de tip cumpărător cere oferte de la un set de agenți de tip vânzător pentru ca apoi să aleagă vânzătorul cu oferta cea mai mică. Schimbul de mesaje este detaliat în figura 6.4.

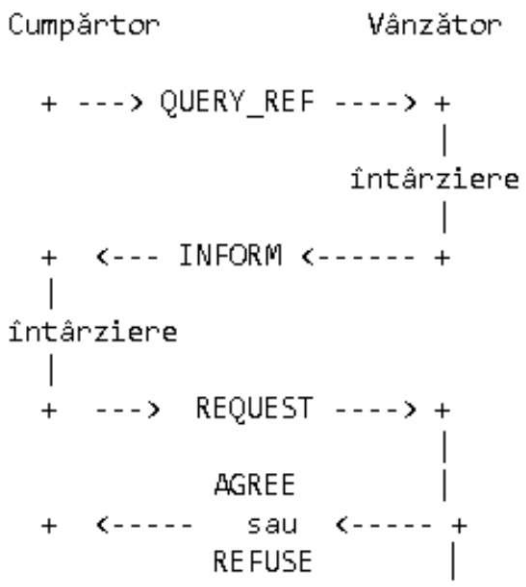

Fig.6.4 – Diagramă secvențială a schimului de mesaje
pentru atribuirea unui produs pe bază de oferte (interacțiune 1 – m)

Pentru implementarea interacțiunii descrise anterior trebuie rezolvate un set de probleme, precum:

A. utilizarea de timpi maximi de recepție (*timeouts*) pentru a limita timpul petrecut în așteptarea răspunsurilor – v. § 5.9, în care s-a dezvoltat un comportament în acest sens, comportament ce poate fi reutilizat în implementarea acesui tip de interacțiune;
B. înlănțuirea comportamentelor pentru a realiza conversații coerente;
C. gestionarea cererilor paralele (o cerere paralelă reprezintă interacțiunea 1–*m*, o conversație paralelă reprezintă interacțiunea *n*–*m*; din anumite rațiuni tehnice – descrise în secțiunea aferentă DataStore cele două cazuri se implementează diferit, cel de-al doilea caz (mai generic) fiind tratat în secțiunea următoare);
D. tratarea mesajelor nedorite rămase în coada de mesaje.

Punctele B și C la comun realizează procesul de recepție paralelă (în același timp) a unei oferte de la mai mulți participanți.

În cadrul acestui material presupunem existența mai multor vânzători identificați prin: „s1“, „s2“ și „s3“. Pentru a determina care oferă cel mai bun preț, cumpărătorul trebuie să interogheze individual fiecare vânzător. Astfel, sunt trimise mesaje de tip `QUERY_REF` tuturor celor trei vânzători și apoi se așteaptă răspunsuri într-un anumit interval de timp (expirarea intervalului de timp presupune faptul că nu există o ofertă de la agentul respectiv). Din comportamentele compozite ale platformei JADE, folosim un comportament secvențial în care primul pas este un comportament paralel care conține trei comportamente de recepție cu timeout, urmat de un al doilea pas/subcomportament în care se ia decizia (se alege agentul cu oferta cea mai bună) și un pas final în care se așteaptă confirmarea ofertei de la vânzător. Secvența de pași descrisă este sintetizată în figura 6.5:

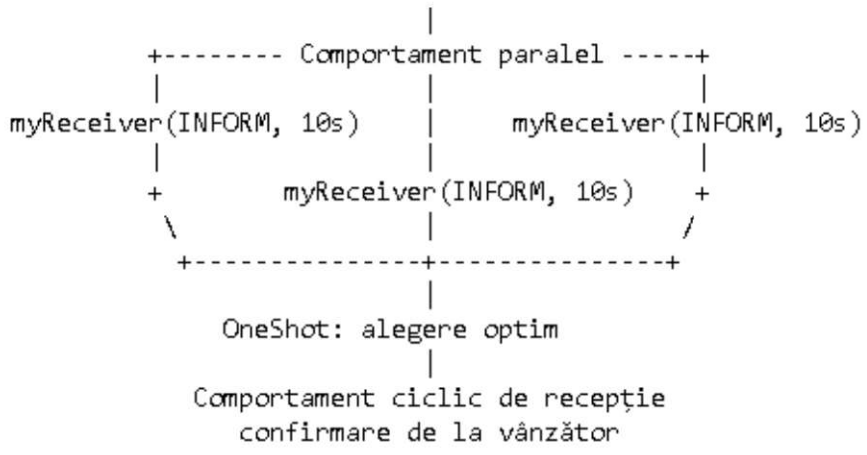

Fig.6.5 – Recepția mai multor oferte în paralel

7. Structura și metoda de lansare în execuție a agenților cumpărător și vânzător

Agentul care oferă serviciul (cel care vinde produsul în cazul protocolului de interacțiune CNP – *Contract Net Protocol* (fig. 6.6)) se mai numește și agent care răspunde (*responder*) sau participant, el așteptând în primă instanță un mesaj pentru a demara execuția. El este compus dintr-un singur comportament ciclic pentru recepția unui mesaj de tip CFP (apel la participare, *Call for Proposal*), urmat de un timp mort pentru simularea unei procesări/luarea unei decizii și apoi se continuă cu răspunsul la mesajul inițial. În cazul interacțiunii 1 (inițiator) – m (participanți) nu se pune problema ca acest tip de agent să interacționeze cu mai mulți inițiatori; de aceea este suficient un comportament de tip ciclic sau, în cazul în care se dorește și o confirmare ulterioară ofertei, se poate extinde comportamentul ciclic la unul secvențial în care în prima etapă se primește un CFP, apoi se așteaptă răspunsul în eventualitatea în care s-a răspuns afirmativ la CFP. Singurul element important pe care trebuie să îl memoreze agentul, element care este comun tuturor comportamentelor sale, este mesajul inițial primit de la inițiatorul interacțiunii (agentul cumpărător). În cazul interacțiunii *n* (inițiatori) – *m* (participanți), situația se complică deoarece agenții participanți intră în interacțiune cu mai mulți inițiatori, aceștia din urmă având posibilitatea să prezinte oferte diferite pentru a-și maximiza profitul. Într-o situație reală de negociere, dacă sunt mai multe cereri vânzătorul crește prețul. Acest lucru modifică comportamentul agentului participant pentru că trebuie lansate în paralel mai multe comportamente care reprezintă direcțiile de negociere, fiecare necesitând propriile variabile decizionale, precum și un set de variabile decizionale comune. Acest aspect, ca și o posibilă soluție, sunt prezentate în secțiunea 9 a acestui capitol (Transmiterea de informații între sub-comportamente – DataStore). O alterantivă la această soluție o reprezintă plasarea metodei care creează conversația într-o clasă ce reprezintă însăși conversația. Pentru fiecare conversație, creăm un obiect de tip conversație distinct și apelăm metoda de configurare a acestuia (*setup*). Atributele obiectului de conversație pot fi acum folosite drept context de către toate comportamentele membre, fiecare interacțiune a agentului de tip vânzător având astfel propriile variabile.

Având în vedere că agentul de tip participant/vânzător trebuie să existe în momentul lansării în execuție a protocolului de intracțiune, se recomandă ca platforma SMA să fie lansată în mai multe etape, la început situându-se configurațiile de rulare ale agentilor participanți și apoi configurațiile de rulare ale agenților inițiatori.

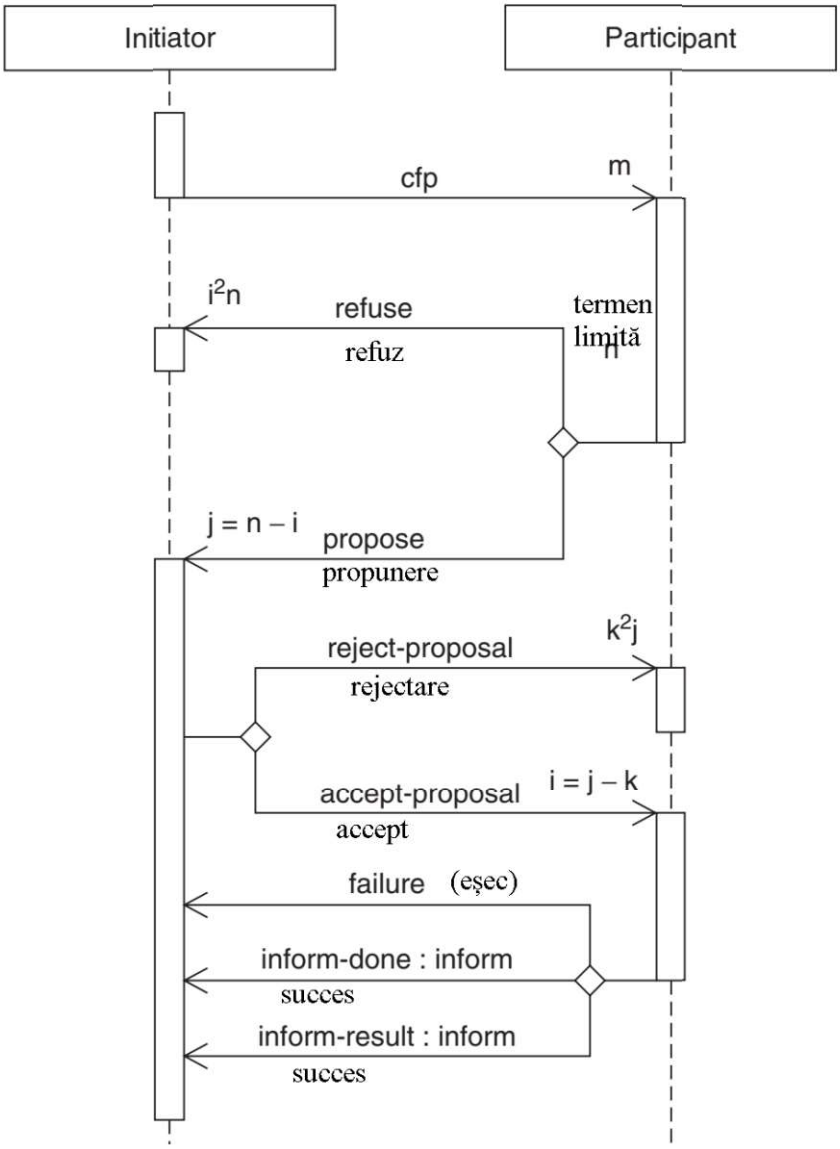

Fig.6.6 – Protocolul de interacțiune FIPA Contract-Net (CNP)

În ceea ce privește agenții inițiatori, aceștia au o structură standard compusă dintr-un comportament secvențial care are ca etape: a) lansarea CFP urmată de b) primirea paralelă a propunerilor (conform Fig.6.5) și la final c) alegerea câștigătorului pe baza celei mai bune oferte.

8. Generarea de identificatori unici pentru agenți

Un aspect important în implementarea protocoalelor de interacțiune îl reprezintă atribuirea unui identificator unic fiecărei conversații (câmpul `conversationID` (CID)). Acesta este un mecanism bine cunoscut: există deja câmpul rezervat pentru CID în toate mesajele ACL, iar șablonul de recepție de mesaje JADE permite selectarea mesajelor pe baza

conversationID. În plus, metoda `createReply` setează CID-ul din răspuns să fie identic cu cel din mesajul original. Folosind același CID pentru toate mesajele schimbate între doi agenți în cursul unei interacțiuni, este relativ banal să se separe acțiunile unei interacțiuni de cele ale unei alte interacțiuni. Conform celor enumerate anterior, principalele elemente ce trebuie realizate în implementarea unui protocol de interacțiune sunt: a) generarea de identificatori de conversație unici (`conversationID`) și b) transmiterea acestor identificatori tuturor comportamentelor aferente agentului care intră în conversația respectivă.

Algoritmii de generare de numere aleatorii au o componentă deterministă astfel încât succesiunea de numere generate depinde în totalitate de numărul dat ca valoare inițială (*seed*). Două generatoare care sunt inițializate cu aceeași valoare vor da exact același set de numere. În situația în care aceste valori aleatorii reprezintă o serie de costuri aferente unor operații propuse, este important ca acestea să fie complet diferite pentru a evita oferte similare. De valori aleatorii (complet diferite) este nevoie și în cazul identificatorilor de conversație pentru ca interacțiunea între doi agenți să nu fie perturbată de alți agenți care fac parte din același sistem multi-agent. În mod implicit, generatoarele Java sunt inițializate din ceasul sistemului, `System.currentTimeMillis`. În general, acest lucru este bun, dar în cazurile precedente, toți agenții sunt generați la începutul execuției și din experimente succesive s-a observat că mulți au fost creați în timpul aceluiași ciclu de ceas, rezultând astfel nume identice, propuneri de prețuri identice, respectiv identificatori de conversație identici. În acest sens, se propune următoarea soluție individuală fiecărui agent, indiferent că aceștia sunt rulați în paralel: inițializare generatoare de numere aleatorii cu o combinație de timp sistem și codul hash al agentului în cauză. În continuare este prezentată metoda propusă pentru a dezvolta generatoare independente care să creeze valori garantat distincte.

```
Random newRandom()
    { return  new Random( hashCode() + System.currentTimeMillis()); }
```

9. Transmiterea de informații între subcomportamente – DataStore

O conversație este o succesiune de comportamente care funcționează împreună pentru realizarea unui scop, partajând informații prin variabile comune. Pentru a permite conversații paralele, trebuie să oferim fiecărei conversații propriul set de variabile comune de comunicare. Din păcate, pentru a simplifica gestionarea memoriei, limbajul Java are anumite

particularități când vine vorba de variabilele globale pe care le pot folosi clasele interne (mesajul primit în comportamentul ciclic care declanșează comportamentul secvențial de interacțiune nu poate fi declarant ca variabilă internă a clasei `CyclicBehaviour`, iar declararea ca variabilă internă a clasei agent va genera rescrierea sa de fiecare dată când se recepționează un nou mesaj). În special, nu se permite să se facă referire la variabilele declarate în metodele ce aparțin părintelui (*surrounding methods*) (Fig.6.7).

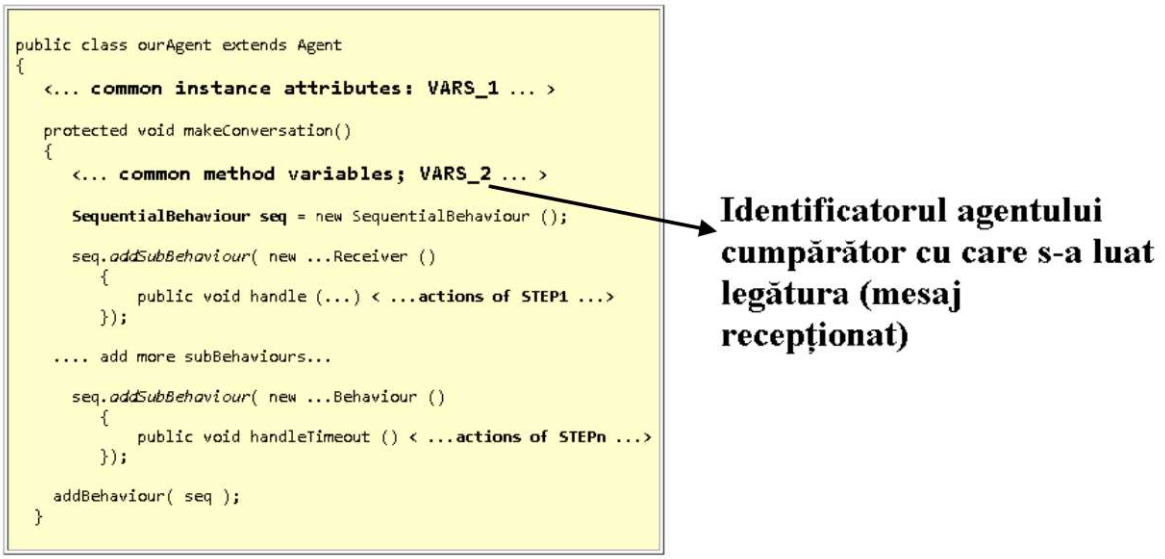

Fig.6.7 – Vizibilitatea variabilelor Java în funcție de locația de declarare

În acest sens, există următoarele soluții pentru a trece peste impedimentul menționat: 1) crearea unei clase de interacțiune (în loc de comportament secvențial) în care transmitem drept constructor mesajul primit, având astfel posibilitatea de a reține toate mesajele recepționate în ciuda faptului că variabila globală de tip `ACLMessage` (din comportamentul ciclic) este rescrisă la fiecare recepție, 2) folosirea `DataStore` ca să partajăm mesaje între comportamente sau 3) realizarea interacțiunilor multiple în mod secvențial; astfel, conversațiile paralele vor deveni succesive, nemaipunându-se problema rescrierii variabilelor globale (e.g.: la primirea unui mesaj se epuizează metodele de interacțiune între agenții implicați și abia apoi se trece la următoarea interacțiune); avantajele acestei metode constau în simplitatea ei, dar în cazul în care comportamentul secvențial aferent durează foarte mult, sau în cazul cel mai defavorabil se blochează, pot apărea probleme; de aceea se recomandă abordarea acestei soluții cu mare atenție.

La realizarea de comportamente de tip secvențial, mașină cu stări finite sau paralel, este de obicei cazul ca un subcomportament să aibă nevoie de acces la unele date produse de alte subcomportamente.

```
public class MySequentialBehaviour extends SequentialBehaviour {
      private ACLMessage receivedMsg;
      public MySequentialBehaviour(Agent a) {
            super(a);
            // . . .
            addSubBehaviour(new SimpleBehaviour(a) {
                  private boolean finished = false;
                  public void action() {
                        receivedMsg = myAgent.receive();
                        if (receivedMsg != null) {
                        finished = true;
                        }
                        else {
                        block();
                        }
                  }
                  public boolean done() {
                  return finished;
                  }
            } );
            addSubBehaviour(new OneShotBehaviour(a) {
                  public void action() {
                  // Process receivedMsg
            }
      } );
      }
}
```

În multe cazuri, este util să se creeze comportamente care pot fi reutilizate în contexte diferite și, prin urmare, nu sunt legate de un anumit agent sau de un comportament compozit părinte dat. În aceste cazuri, datele partajate între comportamente nu pot fi stocate în variabilele membre ale comportamentului agent sau părinte. Clasa `DataStore` inclusă în pachetul `jade.core.behaviours` oferă o soluție simplă și generică la această problemă. Fiecare comportament are propria sa instanță din clasa `DataStore` și prin intermediul metodelor `getDataStore()` și `setDataStore()` din clasa `Behavior` pot fi accesate informațiile stocate în alte comportamente, respectiv se poate stoca informație pentru a fi accesată de alte comportamente. Un obiect de tip `DataStore` este practic o hartă (structura `DataStore` extinde `HashMap`) și oferă mecanismul „standard“ prin care comportamentele concepute pentru a fi reutilizabile trebuie să partajeze date. Adică, setând aceeași instanță `DataStore` între mai multe comportamente, acestea au un spațiu comun în care pot stoca date care trebuie partajate. Astfel, dacă avem un comportament generic de recepție a unui mesaj (`MessageReceiver`) și dorim să-l folosim în pasul *n* al exemplului amintit, putem folosi următoarea secțiune de cod în cadrul comportamentului dorit pentru postarea unui mesaj:

```
public class MessageReceiver extends SimpleBehaviour {
      public static final String RECV_MSG = "received-message";
      private boolean finished = false;

      public void action() {
            ACLMessage msg = myAgent.receive();
            if (msg!= null) {
                  getDataStore().put(RECV_MSG, msg);
                  finished = true;
            }
            else { block();}
      }
      public boolean done() {
      return finished;}
}
```

În mod similar trebuie modificat și codul aferent comportamentului consumator al mesajului:

```
SequentialBehaviour sb = new SequentialBehaviour(anAgent);
Behaviour b = new MessageReceiver(anAgent);
b.setDataStore(sb.getDataStore());
sb.addSubBehaviour(b);
b = new OneShotBehaviour(anAgent) {
public void action() {
ACLMessage receivedMsg = getDataStore().get(MessageReceiver.RECV_MSG);
// Process receivedMsg
}
};
b.setDataStore(sb.getDataStore());
sb.addSubBehaviour(b);
```

10. Protocoale de interacțiune generice

Una dintre componentele cheie ale sistemelor multi-agent este comunicarea. Agenții trebuie să poată comunica cu utilizatorii, cu resursele sistemului și între ei pentru cooperare, colaborare și negociere. Aceștia interacționează între ei prin utilizarea unor limbaje speciale de comunicare, numite limbaje agent de comunicare (*Agent Communication Language* – ACL). În momentul de față, cel mai utilizat limbaj de comunicare între agenți este FIPA-ACL. Caracteristicile principale ale FIPA ACL sunt posibilitatea de a utiliza diferite limbaje de conținut și gestionarea conversațiilor prin protocoale de interacțiune predefinite. Coordonarea este un proces prin care agenții interacționează pentru ca o comunitate de agenți individuali să acționeze într-o manieră unitară. Din motivele pentru care agenții trebuie să fie coordonați, amintim următoarele: (1) scopurile agenților pot genera conflicte între acțiunile agenților, (2) scopurile agenților pot fi interdependente, (3) agenții pot avea capacități și cunoștințe diferite și (4) scopurile agenților pot fi atinse mai rapid dacă agenți diferiți lucrează la fiecare dintre ele. Coordonarea dintre agenți poate fi gestionată cu o varietate de abordări, inclusiv structurarea organizațională, contractarea, planificarea și negocierea cu mai mulți agenți.

FIPA specifică un set de protocoale de interacțiune standard, care pot fi folosite ca șabloane pentru dezvoltarea de conversații complexe în sistemele multi-agent. Pentru fiecare conversație între agenți, JADE definește rolul de inițiator (agentul care începe conversația) și rolul de participant (*responder*), agentul care se angajează într-o conversație după ce a fost contactat de un agent inițiator. JADE oferă clase care implementează comportamente gata făcute pentru ambele roluri din conversație pentru majoritatea protocoalelor de interacțiune FIPA. Aceste clase pot fi găsite în pachetul `jade.proto`. Ele oferă un set de metode virtuale pentru a gestiona stările protocoalelor.

Toate clasele care oferă suport pentru implementarea protocoalelor standard de interacțiune în JADE sunt incluse în pachetul `jade.proto`. Atunci când participă la o conversație conform unui protocol de interacțiune, un agent implementează fie rolul de inițiator, fie cel de participant. În consecință, clasele din pachetul `jade.proto` sunt împărțite în inițiatori și participanți (*responders*). De exemplu, avem `ContractNetInitiator` și `ContractNetResponder`, `SubscriptionInitiator` și `SubscriptionResponder` și acest principiu se regăsește și la celelalte protocoale de interacțiune. A juca un rol într-o conversație, indiferent dacă este rolul de inițiator sau de participant, implică executarea unei sarcini de un anumit fel și astfel toate clasele protocolului de interacțiune (atât inițiatorii, cât și cei care răspund – participanții) sunt comportamente JADE.

Toate comportamentele aferente agentului Inițiator se încheie și sunt eliminate din coada sarcinilor agentului de îndată ce ajung la orice stare finală a protocolului de interacțiune. Pentru a permite reutilizarea obiectelor Java care reprezintă aceste comportamente fără a fi nevoie să se recreeze noi obiecte, toți inițiatorii includ o serie de metode de resetare cu argumentele adecvate (comportament.reset). În plus, toate comportamentele inițiatorului, cu excepția `FipaRequestInitiatorBehaviour`, sunt 1:N, adică pot gestiona mai mulți agenți de tip *responder* în același timp. Toți constructorii clasei comportamentului inițiator conțin un parametru de tip `ACLMessage` care reprezintă mesajul utilizat pentru a iniția protocolul. De exemplu, clasa `ContractNetInitiator` primește mesajul CFP care urmează a fi trimis participanților pentru a iniția procedura de delegare a sarcinilor. Toate clasele care implementează inițiatorii de protocol acceptă atât interacțiuni unu-la-unu (*one-to-one*), cât și unu-la-mulți (*one-to-many*), în funcție de numărul de receptori specificat în mesajul de inițiere.

Toate comportamentele ciclice ale agentului participant sunt reprogramate de îndată ce ajung la orice stare finală a protocolului de interacțiune. Astfel, dezvoltatorul poate limita numărul maxim de comportamente de răspuns pe care agentul trebuie să le execute în paralel. De exemplu, următorul cod asigură că maximum două sarcini de tip CONTRACT-NET (protocol de interacține FIPA) vor fi executate simultan:

```
Protected void setup() {
addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
}
```

Versiunea ciclică are un parametru `MessageTemplate` în constructorul său, folosit pentru a selecta mesajele de inițiere a protocolului de la inițiatori. Un comportament de răspuns al protocolului ciclic este de obicei adăugat în configurarea agentului și rămâne activ pe toată durata de viață a agentului. De fiecare dată când este primit un mesaj de inițiere a protocolului care se potrivește cu șablonul, comportamentul de răspuns la protocol îl procesează, realizează conversația și apoi revine în starea de așteptare a unui nou mesaj de inițiere. Astfel, atunci când se utilizează versiunea ciclică a unui comportament de răspuns, un agent poate fi angajat într-o singură conversație condusă de acel protocol la un moment dat. De exemplu, dacă două mesaje CFP sunt primite de un agent care rulează un comportament `ContractNetResponder`, acesta din urmă va finaliza conversația inițiată de primul CFP înainte de a-l servi pe al doilea.

În tabelul 6.1 se regăsesc protocoalele de interacțiune conform standardului FIPA care au o implementare în JADE, versiunea 4.5.

Tabel 6.1 – Protocoale de interacțiune FIPA implementate în JADE , versiunea 4.5

Protocol	**Clasa initiator**	**Clasa participant**
FIPA-Request	AchieveREInitiator	AchieveREResponder
FIPA-Query		
FIPA-Propose	ProposeInitiator	ProposeResponder
Versiunile iterative ale	IteratedAchieveREInitiator	SSIteratedAchieveREResponder
FIPA-Request		
FIPA-Query		
Contract-Net	ContractNetInitiator	ContractNetResponder
		SSContractNetResponder
FIPA-Subscribe	SubscriptionInitiator	SubscriptionResponder

Clasele care implementează participanții la un protocol de interacțiune oferă o serie de metode (*callback methods*) pe care programatorii trebuie să le redefinească prin implementarea unei logici dependente de aplicația pe care o dezvoltă (în protocolul CNP în starea de decizie ce urmează apelului CFP se alege agentul care are oferta cea mai bună). Toate aceste metode sunt declarate protejate și au o implementare implicită (de obicei, goală). Numele metodelor este `handlcXXX`, unde porțiunea `XXX` reprezintă etapa aferentă protocolului de interacțiune (de exemplu: în protocolul CNP (Fig.6.6), secțiunea participant, etapa aferentă procesării tuturor mesajelor de tip apel la participare (CFP) se numește `handleCfp`). În mod similiar sunt denumite și celelalte puncte de intrare în procedurile de procesare (metode de tip *callback*). O detaliere a numelor acestora, precum și a parametrilor necesari apelării lor, rezultă din analiza protocolului conform standardului FIPA (www.fipa.org) sau, alternativ, se poate consulta implementarea protocolului în sursele JADE (`JADE-src-\jade\src \jade\proto`) și extrage numele procedurii ce trebuie redefinită. Acest nume se poate extrage din comentariile aferente implementării sau consultând codul și identificând procedurile aferente, acestea având o implementare vidă. În acest fel programatorii pot alege, în funcție de cerințele lor specifice, ce metode să implementeze și pe care să le ignore. Atât pentru agenții de tip inițiator, cât și pentru cei de tip participant, majoritatea metodelor de tip callback sunt invocate în urma primirii unui mesaj și au următorul format:

```
protected handle<message-performative>(ACLMessage receivedMessage)
```

Un exemplu este clasa `ContractNetResponder`, în instanța căreia dacă este primit un mesaj de tip ACCEPT_PROPOSAL este apelată metoda `handleAcceptProposal` având ca parametru mesajul primit. Atunci când recepția unui mesaj încheie o interacțiune cu expeditorul acelui mesaj (e.g.: când un mesaj de tipul REFUSE este primit ca răspuns la un mesaj de tip CFP într-un protocol Contract-Net se indică faptul că nu mai trebuie trimis niciun mesaj înapoi celui care răspunde), metoda corespunzătoare `handleXXX()` returnează o valoare nulă. Pe de altă parte, dacă un răspuns trebuie trimis înapoi, distingem două cazuri:

a) pentru participanții/respondenții care sunt întotdeauna implicați în interacțiuni unu-la-unu, metoda `handleXXX()` returnează un obiect de tipul `ACLMessage`. Valoarea returnată va fi folosită ca răspuns. De exemplu, metoda `handleCfp()` a `ContractNetResponder` va fi de obicei redefinită după cum urmează:

```
protected ACLMessage handleCfp(ACLMessage cfp) {
ACLMessage reply = cfp.createReply();
// Evaluarea apelului
if (call OK) {
// Pregătirea unei propuneri
reply.setPerformative(ACLMessage.PROPOSE);
}
else {
reply.setPerformative(ACLMessage.REFUSE);
}
return reply;
}
```

b) pentru agenții de tip inițiator care sunt proiectați să implementeze interacțiuni multiple (de tip *one-to-many*), metoda `handleXXX()` primește un argument suplimentar de tip vector la care trebuie adăugat răspunsul. De exemplu, metoda `handlePropose()` a `ContractNetInitiator` va fi de obicei redefinită după cum urmează:

```
protected void handlePropose(ACLMessage propose, Vector acceptances)
{
ACLMessage reply = propose.createReply();
// Evaluarca propuncrii
if (proposal OK) {
reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
}
else {
reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
}
acceptances.add(reply);
}
```

Pe lângă metodele de tip callback executate la recepția mesajelor aferente protocolului de interacțiune, clasele din pachetul `jade.proto` oferă alte două tipuri de metode callback atunci când este cazul. Inițiatorii proiectați pentru a implementa interacțiuni de tipul one-to-many oferă metode care sunt invocate atunci când au fost colectate răspunsurile de la toți participanții/respondenții. Aceste metode au forma `handleAllXXX(Vector v)` și permit tratarea tuturor răspunsurilor în același timp. De exemplu, clasa `ContractNetInitiator` furnizează metodele `handleAllResponses()` și `handleAllResultNotifications()`. Acestea sunt invocate atunci când sunt primite răspunsuri (de exemplu, PROPOSE/REFUSE/NOT_UNDERSTOOD) de la toți participanții și, respectiv, notificări de rezultat (adică INFORM/FAILURE) de la participanții ale căror propuneri au fost acceptate.

În unele cazuri, o clasă care implementează un protocol de interacțiune trebuie să trimită unul sau mai multe mesaje în urma unui eveniment care nu are legătură directă cu

recepția altui mesaj. De exemplu, într-un protocol FIPA-Request, dacă un participant a răspuns cu un AGREE la un mesaj de REQUEST primit, acesta trebuie să trimită succesiv un INFORM sau un FAILURE pentru a notifica solicitantul despre rezultatul acțiunii convenite. Aceste cazuri sunt implementate prin metode de tip callback care iau forma `prepareXXX()` și returnează fie un obiect de tipul `ACLMessage`, fie un vector de mesaje. De exemplu, clasa `AchieveREResponder` (care implementează rolul de participant în protocolul FIPA-Request) oferă metoda `prepareResultNotification()` pentru a acoperi cazul din exemplul discutat.

Metoda `prepareCFPs()` este apelată de îndată ce începe comportamentul `ContractNetInitiator`. Ea are rolul de pregătire a mesajelor CFP care urmează să fie trimise către participanți, fiind deosebit de utilă atunci când mesajul CFP nu este cunoscut la inițializare sau când este necesar să trimitem mesaje personalizate fiecărui participant. Toate clasele de inițiatori ai unui protocol de interacțiune au metode similare.

Capitolul 7: Realizarea unei interfețe grafice pentru agenții JADE

Cuprins

1. Introducere

După cum s-a văzut în capitolul anterior, depanarea unui protocol de interacțiune în care sunt implicați mai mulți agenți de tip inițiator și participant necesită o serie de configurații de rulare pentru a separa cele două capete ale comunicației, pentru ca ulterior să se pornească elementul de monitorizare (agentul de tip Sniffer). Mai mult, în lucrul cu consola este dificil să se identifice corect mesajele tipărite în cazul în care sunt lansați la comun mai mulți agenți printr-o singură configurație de rulare. În acest context, adăugarea unei/unor interfețe grafice (*Graphical User Interface* – GUI) asociate în mod individual fiecărui agent simplifică depanarea proiectelor complexe.

În limbajul de programare Java, o interfață grafică rulează în propriul fir de execuție (firul de gestiune a evenimentelor), care îi permite să gestioneze și să reacționeze prompt la evenimentele generate de fiecare dată când utilizatorul interacționează cu ea printr-o componentă grafică, precum apăsarea unui buton sau redimensionarea ferestrei curente. În același timp, un agent software rulează în propriul fir de execuție care îi permite să-și gestioneze comportamentele. Deoarece pot apărea erori la apelarea de metode între fire de execuție, JADE oferă un mecanism adecvat pentru a gestiona interacțiunile dintre cele două fire atunci când se integrează o interfață grafică cu un agent. Acest mecanism se bazează pe

transmiterea evenimentelor de interes (pentru sincronizare), evenimente care se transmit în două direcții: a) de la agent la interfață (fluxul de la dreapta la stânga din figura 7.1), constând în actualizări ca urmare a unor procese asociate agentului (de exemplu: actualizarea unui câmp de tip text cu conținutul mesajului primit de agent) și b) de la interfață la agent (fluxul de la stânga la dreapta din figura 7.1) ca urmare a transmiterii unui eveniment captat de interfață, eveniment care este de interes pentru agent (de exemplu: apăsarea unui buton poate declanșa execuția unui comportament sau lansarea unui mesaj, care ulterior declanșează un protocol de interacțiune). O sinteză a celor două interacțiuni, precum și a componentelor Java necesare implementării interacțiunii agent-interfață este prezentată grafic în figura 7.1.

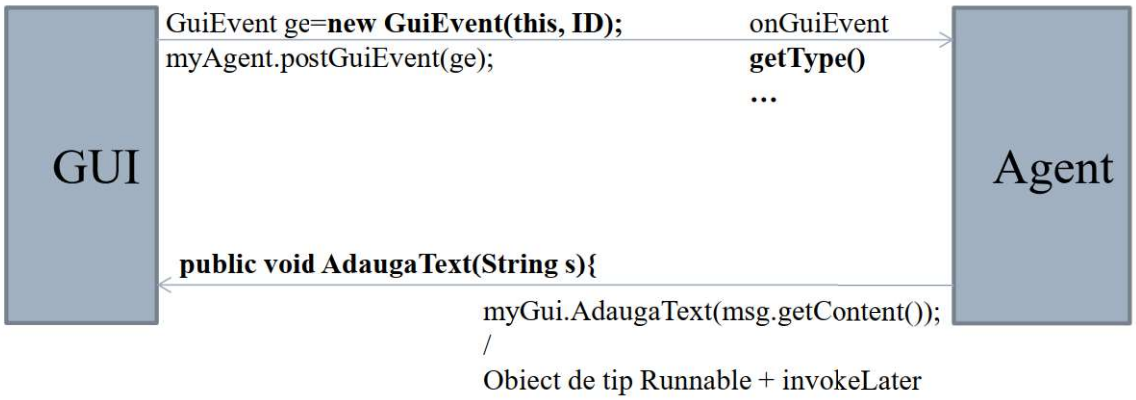

Fig.7.1. – Interacțiunea agent-interfață grafică

O problemă tipică pe care dezvoltatorii trebuie să o gestioneze este modul în care agenții JADE trebuie să interacționeze cu GUI (interfața grafică a utilizatorului), și invers. Problema este că trebuie folosite modele de programare a comunicațiilor între fire de execuție. Astfel, firul de execuție al agentului trebuie activat ori de câte ori este primit un mesaj de tip ACL și firul de procesare a evenimentelor (GUI) se activează ori de câte ori componentele grafice declanșează diferite tipuri de evenimente, precum apăsarea unui buton, redimensionarea ferestrei sau glisarea unui control de tip bară (*scrollbar*).

Problemele tipice care trebuie evitate sunt reacția la un eveniment grafic prin blocarea firului de procesare a evenimentelor până la primirea unui mesaj ACL sau modificarea variabilelor nesincronizate din ambele fire. În cele ce urmează sunt prezentate o serie de practici recomandate în dezvoltarea unor agenți JADE cu interfețe grafice pentru a evita problemele descrise anterior. Exemplificarea se va face folosind mediul de dezvoltare integrat Eclipse împreună cu unealta de tip plugin WindowBuilder pentru proiectarea și dezvoltarea de interfețe bidirecționale.

2. Interacțiunea interfață grafică->agent

În versiunea de bază, JADE pune la dispoziția dezvoltatorului clasa abstractă `GuiAgent` care extinde clasa `Agent`. Această clasă are două metode specifice: `postGuiEvent()` și `onGuiEvent()`. Acestea sunt cele două metode care permit gestionarea interacțiunilor dintre o interfață grafică și din agentul software asociat. Astfel, pentru a putea utiliza aceste metode, programul dezvoltat trebuie să extindă clasa `GuiAgent`. Apoi se stabilesc informațiile de sincronizat între agent și interfața grafică. Aceste informații sunt codificate în cadrul metodei `onGuiEvent()`, pe care agentul implementat le va utiliza pentru a primi și procesa evenimente care sunt postate de GUI prin metoda `postGuiEvent()` (Fig.7.1). Astfel, metoda `onGuiEvent()` poate fi asemănată metodei `actionPerformed()` din interfața grafică. Când pornește un agent software care extinde clasa `GuiAgent`, acesta lansează un comportament specific – `GuiHandlerBehaviour` – care gestionează evenimentele provenite de la interfața grafică și le trimite către procedurile corespunzătoare. Pentru a transmite un eveniment agentului, GUI creează pur și simplu un obiect `GuiEvent`, adaugă parametrii necesari și îl transmite în argument metodei `postGuiEvent()`. Deoarece această metodă aparține clasei `GuiAgent`, trebuie furnizată interfeței grafice o referință la clasa agent prin care interfața poate invoca acea metodă.

Când agentul primește un eveniment postat de interfață, acesta testează tipul evenimentului (tipul evenimentului reprezintă informația de la GUI ce trebuie sincronizată cu agentul asociat) apelând metoda `getType()`, ce are ca parametru un obiect de tipul `GuiEvent` și, în funcție de identificatorul rezultat, se apelează procedura corespunzătoare. Acest lucru necesită definirea tipurilor de evenimente pe care agentul trebuie să le primească din GUI. Evenimentele trebuie să fie constante întregi.

Tabel 7.1. – Secvență de instrucțiuni pentru transmiterea evenimentului captat în interfață în codul agentului aferent

Cod interfață grafică	Cod agent

`class InterfataAgent extends Jframe`
`{`
`...`
`//myAgent - referință agent părinte`
`//metodă tip ActionListener aferentă controlului care`
	`public class Agent_cu_interfata extends GuiAgent`
	`{`
	`...`
	`InterfataAgent my_local_interf = new InterfataAgent (this);`
	`my_local_interf.setVisible(true);`
	`...`

declanșează evenimentul în intrefață
GuiEvent ge = new GuiEvent(this, COD_EVENIMENT);
myAgent.postGuiEvent(ge);
...
}
	protected void onGuiEvent(GuiEvent ev) {
	eveniment = ev.getType();
	//structura CASE pentru procesarea unui anumit eveniment
	...
	}

O alternativă în transmiterea de mesaje de la interfață la agent constă în crearea de comportamente și planificarea execuției lor în firul de execuție al agentului. Când un agent are o interfață grafică, de obicei trebuie să reacționeze la acțiunile utilizatorului, cum ar fi pornirea unei noi conversații atunci când utilizatorul apasă un buton. Când apare un eveniment de la interfața grafică, metoda `actionPerformed()` este apelată de firul de procesare a evenimentelor pe metoda de tip `ActionListener` asociată cu acea sursă a evenimentului. În cadrul acestei metode, o bună practică de programare este de a pregăti un obiect JADE de tip comportament (Behavior) și de a-l programa pentru execuție în firul agentului.

Codul următor reprezintă o parte din structura agentului JADE RMA (interfața grafică de monitorizaare a platformei) (clasa `jade.tools.rma.rma`). Această metodă este invocată atunci când utilizatorul interacționează cu GUI și selectează terminarea unui agent. Metoda arată cum este instanțiat un comportament cu tot cu argumentele aferente și apoi cum este programat pentru execuție.

```
public void actionPerformed(ActionEvent e) {
/*omis*/
AgentTree.Node curNode =
(AgentTree.Node)panel.treeAgent.tree
.getSelectionPath();
rma.killAgent(new AID(curNode.getName(), AID.ISLOCALNAME));
/*omissis*/
}
public void killAgent(AID name) {
KillAgent ka = new KillAgent();
ka.setAgent(name);
try {
Action a = new Action();
a.setActor(getAMS());
a.setAction(ka);
ACLMessage requestMsg = getRequest();
```

```
requestMsg.setOntology(JADEManagementOntology.NAME);
getContentManager().fillContent(requestMsg, a);
addBehaviour(new AMSClientBehaviour("KillAgent", requestMsg));
} catch(Exception fe) {
fe.printStackTrace();
}
}
```

Este de reținut că obiectele de tip comportament sunt programate pentru execuție numai după ce metoda `setup()` a agentului părinte s-a încheiat. Comportamentele sunt întotdeauna executate în firul agentului. În consecință, nu este necesară nicio sincronizare între comportamentul agent și cel aferent interfeței grafice. Desigur, în unele cazuri, adăugarea unui comportament poate fi un proces anevoios și chiar inutil atunci când reacția la o acțiune grafică poate fi pur și simplu modificarea valorii unei variabile sau pregătirea și trimiterea unui mesaj `ACLMessage` (trimiterea unui mesaj este un proces asincron). Toate acestea sunt activități valide pentru firul interfeței grafice și, în general, nu provoacă nicio problemă. În schimb, blocarea apelurilor (de exemplu, `Agent.blockingReceive ()`) nu trebuie executată niciodată în cadrul firului interfeței grafice.

3. Interacțiunea agent->interfață grafică

O interfață grafică are un mecanism încorporat de gestionare a evenimentelor, mecanism care este implementat prin metoda `actionPerformed()` a fiecărei componente care este înregistrată cu un obiect `ActionListener`. Pentru a înregistra o componentă a interfeței grafice cu un obiect `ActionListener`, fie se procedează astfel încât interfața grafică să implementeze metoda `ActionListener` și apoi se înregistrează toate componentele interactive ale GUI, cum ar fi butoanele, cu acest `ActionListener` prin metoda `addActionListener`; fie, pentru fiecare dintre componentele interactive se creează în mod anonim un obiect `ActionListener` care este adăugat la componentă trecându-l în argument la aceeași metodă `addActionListener()`. Ori de câte ori se face un apel către GUI, un `ActionEvent` este generat de componenta sursă, care invocă metoda `actionPerformed()`. Ca urmare, conform codului furnizat în cadrul metodei `actionPerformed()`, GUI răspunde prin procesarea evenimentului. Când agentul software interacționează cu GUI, acesta apelează doar metoda definită în cadrul programului GUI care activează acest mecanism.

O alternativă, mai corectă din punctul de vedere al comunicației între firele de execuție, este utilizarea unui obiect de tip `Runnable` care este instanțiat în codul (firul de execuție) agent şi inserat în coada de procesare a evenimentelor aferentă firului de execuție al interfeței grafice. Deoarece agentul are propriul fir de execuție, se poate deduce imediat că actualizarea GUI din acest fir poate duce la probleme neașteptate din cauza problemelor de sincronizare. Uneltele AWT, Swing, precum şi majoritatea cadrelor de dezvoltare de interfețe grafice oferă o metodă ad-hoc adecvată care pune în coada aferentă interfeței grafice un obiect Runnable şi determină executarea lui asincronă pe firul de distribuire a evenimentelor GUI:

- java.awt.EventQueue.invokeLater() pentru AWT;
- javax.swing.SwingUtilities.invokeLater() pentru Swing;

Prin urmare, recomandarea este de a încapsula într-un obiect Runnable toate accesările obiectelor interfeței grafice dintr-un comportament JADE şi, în general, dintr-un fir care nu este `EventDispatchThread`. Apoi, trebuie utilizată metoda adecvată pentru a trimite acest obiect `Runnable` la `EventDispatchThread`.

Tabel 7.2. – Secvență instrucțiuni transmitere a evenimentului captat în agent în codul interfeței grafice aferente agentului

Cod agent	Cod interfață grafică

`public class Agent_cu_interfata extends GuiAgent`
`{`
`...`
`InterfataAgent my_local_interf = new InterfataAgent (this);`
`my_local_interf.setVisible(true);`
`...`
`my_local_interf.alertResponse()`
 	
`...`
`}`
	`class InterfataAgent extends Jframe`
	`{`
	`...`
	`//myAgent - referință agent părinte`
	`//metodă tip ActionListener aferentă controlului care declanșează evenimentul în intrefață`
	
	`public void alertResponse(Object o) {`
	`//cod interfață grafiă aferent evenimentului agent`
	`}`

Fragmentul de cod care urmează face parte din agentul JADE RMA (jade.tools.rma). Această metodă este invocată atunci când Agentul RMA primește un mesaj prin care se informează că un agent nou a fost creat pe un container dat. Metoda arată cum firul agent

creează o instanță de obiect Runnable și îl trimite la EventDispatchThread al interfeței grafice. Acest obiect Runnable este responsabil pentru actualizarea interfeței grafice prin crearea unui nou obiect javax.swing.tree.TreeNode și adăugarea acestuia la JTree a containerului dat.

```
public void addAgent(final String containerName, final AID agentID)
{
Runnable addIt = new Runnable() {
public void run() {
String agentName = agentID.getName();
AgentTree.Node node = tree.treeAgent.createNewNode(agentName,
1);
/* [cod omis] */
tree.treeAgent.addAgentNode((AgentTree.AgentNode)node,
containerName, agentName, agentAddresses, "FIPAAGENT");
}
};
SwingUtilities.invokeLater(addIt);}
```

4. Utilizarea WindowBuilder pentru generarea unei interfețe grafice

a. Descrierea uneltei WindowBuilder

Realizarea unei interfețe grafice în modul text este un proces complicat și de aceea se recomandă folosirea unei unelte suplimentare precum plugin-ul Eclipse WindowBuilder (https://www.eclipse.org/windowbuilder). Această unealtă permite devoltarea de interfețe grafice folosind metode de tip Drag-and-Drop și asigură sincronizarea permanentă a codului sursă cu interfața grafică. WindowBuilder este compus din SWT Designer și Swing Designer și facilitează crearea de interfețe grafice Java fără alocarea unui timp excesiv procesului de scriere de cod. Se pot folosi modul vizual și instrumentele de tip WYSIWYG (*What-You-See-Is-What-You-Get*) pentru a crea forme grafice simple pentru ferestre complexe, iar codul Java va fi generat și sincronizat în mod automat. Plugin-ul permite adăugarea cu ușurință de controale utilizând opțiunea de glisare și plasare (*Drag-and-Drop*), adăugarea de controale de evenimente (*handlers*) elementelor vizuale, modificarea a diferite proprietăți ale controalelor folosind un editor de proprietăți, precum și internaționalizarea aplicației (proiectarea unei aplicații software astfel încât să poată fi adaptată la diferite limbi și regiuni fără modificări în cadrul codului – acest proces este realizat prin folosirea unor fișiere de configurare care sunt editate în funcție de utilizatorul final).

WindowBuilder este construit ca un plugin pentru Eclipse și diferitele IDE-uri bazate pe Eclipse (RAD, RSA, MyEclipse, JBuilder etc.). Pluginul creează un arbore de sintaxă abstractă (*abstract syntax tree* – AST) pentru a naviga prin codul sursă și folosește GEF (*Graphical Editing Framework* – proiect Eclipse care oferă componente grafice la nivel de utilizator final cât și la nivel de cadre de dezvoltare) pentru a afișa și gestiona prezentarea vizuală.

Codul generat nu necesită biblioteci personalizate suplimentare pentru compilare și rulare: tot codul generat poate fi folosit fără a avea instalat WindowBuilder. WindowBuilder poate citi și scrie aproape orice format și poate realiza ingineria inversă a celor mai multe coduri de interfețe grafice Java scrise manual. De asemenea, acceptă editarea codului în formă liberă (modificările pot fi făcute oriunde, nu doar în zone speciale), precum și mutarea și redenumirea. Editorul grafic este compus din următoarele componente:

- perspectiva de realizare grafică a interfeței (*DesignView*);
- perspectiva de realizare a interfeței folosind cod sursă (*Source View*);
- perspectiva cu componentele proiectului dezvoltat (*Component Tree*) – prezintă grafic relațiile ierarhice între componentele proiectului; de exemplu: pentru o redimensionare corectă a ferestrei principale se recomandă ca toate componentele grafice să fie subordonate unei componente de tip Layout care să aibă un comportament de tip absolut;
- perspectiva cu proprietățile componentelor proiect (*Proprety Pane*) – prezintă proprietățile și evenimentele asociate componentei selectate;
- paletă componentă (*Palette*) – meniu prin care se poate avea acces rapid la componente grafice;
- meniu comenzi rapide uzuale (*Toolbar & ContextMenu*).

Aceste componente sunt prezentate grafic în figura 7.2.

Unealta WindowBuilder oferă următoarele caracteristici importante:

- generare bidirecțională de cod/unelte grafice prin selectarea perspectivei de lucru: Source/Design (fig.7.2);
- internaționalizare – poate externaliza texte, poate crea și gestiona pachete de resurse pentru mai multe limbi, poate schimba locațiile rapid și edita texte în diferite regimuri de funcționare;

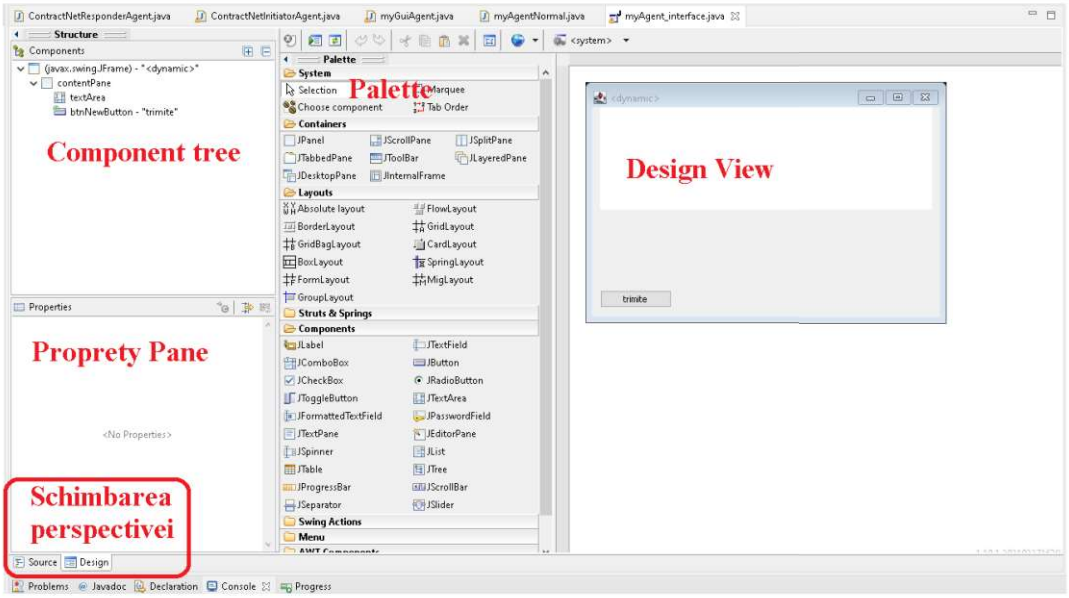

Fig.7.2. – Structura uneltei WindowBuilder

- creare de componente personalizate reutilizabile – unealta permite crearea de componente compozite (SWT) și paneluri JPanel (AWT);
- creare de componente generice versionabile (*Factory method pattern*);
- moștenire a componentelor vizuale;
- adăugare de evenimente – evenimentele apar atunci când utilizatorul interacționează cu interfața grafică. Codul adecvat de gestionare a evenimentelor este apoi executat. Pentru a ști când au loc evenimente, mai întâi trebuie adăugate componentele de gestionare a lor. Unealta WindowBuilder ușurează adăugarea și eliminarea elementelor de monitorizare a componentelor (*event listener*);
- creare și editare vizuală a meniurilor aferente ferestrei GUI;
- conversia între componente similare – editorul permite transformarea de componente similare de la un tip la altul. Când o componentă este transformată dintr-un tip într-altul, se păstrează proprietățile care sunt aceleași între cele două tipuri. Acest lucru permite modificări rapide de structură fără a fi nevoie să se recreeze toate componentele.

Pașii pentru instalarea corectă a plugin-ului WindowBuilder sunt următorii: a) accesarea site-ului principal al proiectului (Fig.7.3) și preluarea link-ului de instalare, b) accesarea meniului de instalare de noi plugin-uri din Eclipse (Fig.7.4) și c) configurarea modulului de instalare a plugin-ului (Fig.7.5).

Fig.7.3. – Site-ul proiectului WindowBuilder

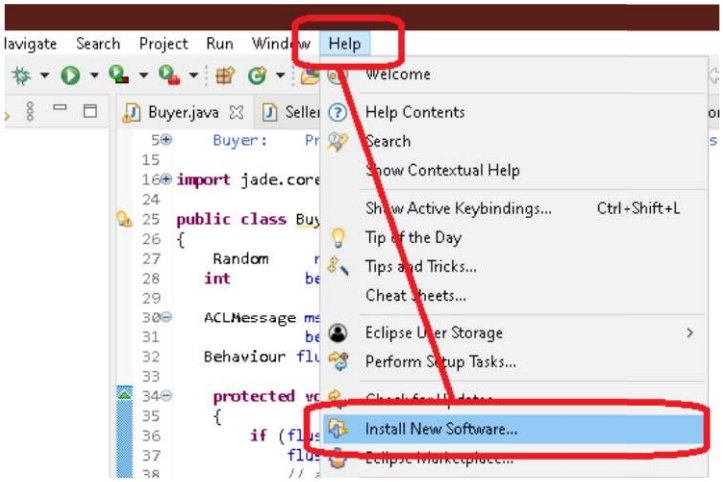

Fig.7.4. – Modalitatea de instalare a unui plugin în Eclipse

b. Realizarea unei componente grafice

Pentru crearea unei componente folosind WindowBuilder se selectează cu click dreapta proiectul în care se dorește adăugarea și apoi New=>Other=>WindowBuilder. Din meniul rezultat se recomandă folosirea componentelor Swing (Swing Designer=>JFrame). Noua componentă rezultată trebuie să arate ca în figura 7.2. Pe această planșetă goală se realizează (recomandat vizual, prin Drag&Drop) interfața dorită. Având în vedere că în mod implicit aspectul (*layout*) ferestrei este cu ancorare pe laturi, respectiv pe centru, se recomandă folosirea unui aspect de tip absolut (*absolute layout*) pentru menținerea dimensiunii dorite a componentelor grafice. Dacă se dorește redesenarea componentelor la modificarea dimensiunii ferestrei, atunci trebuie realizată o combinație de componente de tip *Layout*. Principalele componente grafice folosite în cazul unui proiect multi-agent sunt de tip buton (JButton din Palette=>Components), zonă de text (JtextArea din Palette=>Components), aspect (AbsoluteLayout din Palette=>Layouts) și control de glisare (JscrollPane din Palette=>Containers).

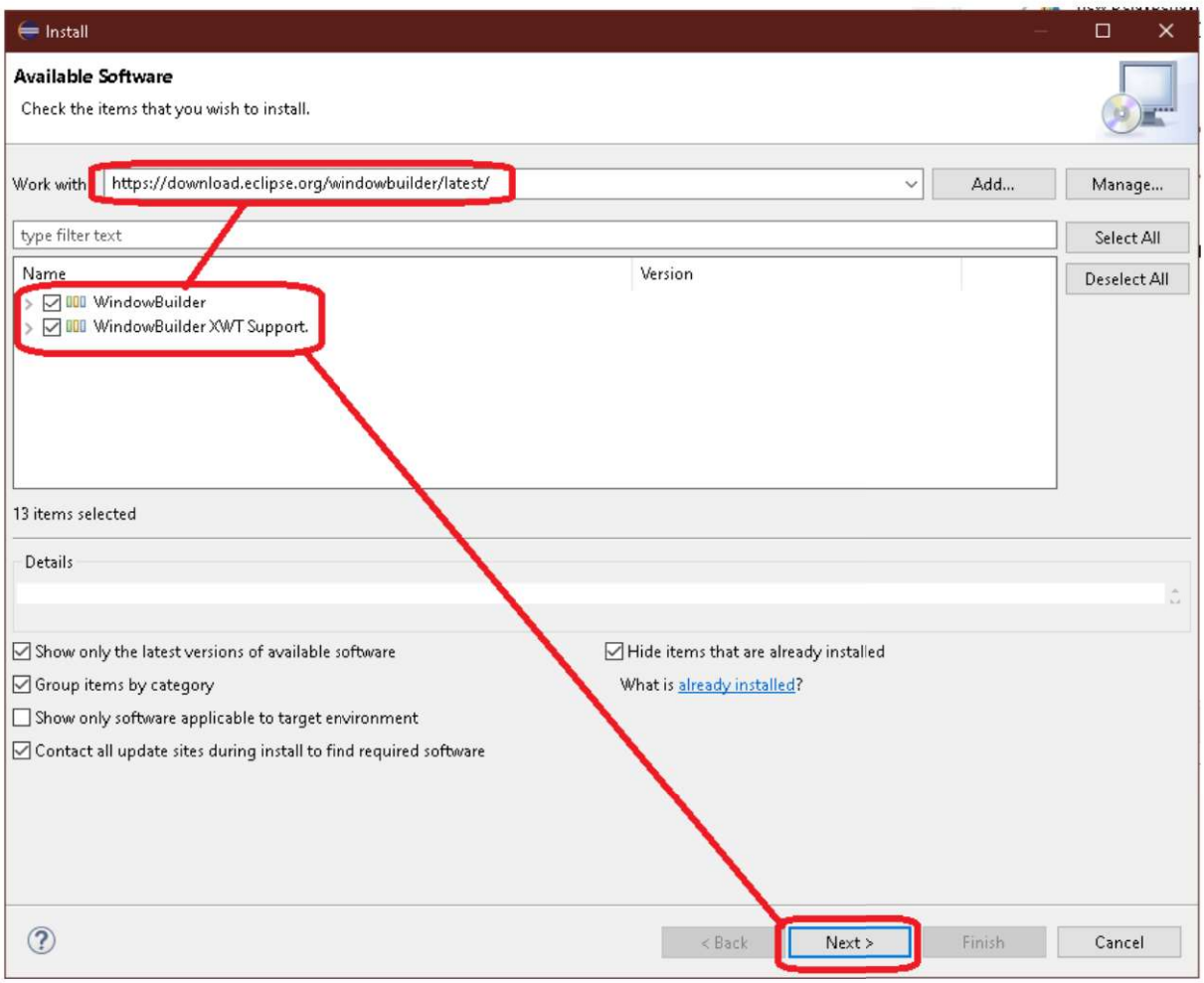

Fig.7.5. – Configurarea instalării plugin-ului WindowBuilder

Acestea au un comportament simplu; de exemplu: a) adăugarea unei metode de tratare a apăsării butonului se face prin dublu click pe buton sau afișând toate evenimentele asociate și selectând din acestea evenimentul dorit, tot cu dublu click; rezultatul este generarea automată a codului de tratare a evenimentului; sau b) modificarea poziției de ancorare se face folosind aspectul absolut; sau c) glisarea textului în situația în care controlul aferent se umple, se realizează prin adăugarea unui element de glisare și apoi adăugarea în el a controlului de text.

c. Integrarea componentei grafice cu codul agent

Fereastra grafică de sine stătătoare nu interacționează în mod implicit cu agentul. Realizarea acestui proces se face prin instanțierea interfeței grafice în codul agentului și alterarea constructorului interfeței astfel încât aceasta să aibă o referință la agent. Respectiva referință va fi folosită pentru situația în care se dorește transmiterea de informații către agent prin postarea de evenimente de tip `GuiEvent`:

```
btnNewButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        GuiEvent ge = new GuiEvent(this, 1);
                        localAgent.postGuiEvent(ge);
                }
        });
```

De exemplu, dacă se dorește ca la apăsarea unui buton să se declanșeze un eveniment în cadrul agentului, atunci acest eveniment va fi codificat printr-un număr întreg. Acest număr va fi transmis din firul de execuție al interfeței folosind codul precedent. Ambele procese sunt descrise detaliat în tabelul 7.1.

Având în vedere că un proiect multi-agent are mai mulți agenți, diferiți sau de același tip (mai multe instanțe cu nume diferite), în situația în care acești agenți au interfețe grafice asociate trebuie realizată o diferențiere între aceștia. În acest sens, se recomandă modificarea suplimentară a constructorului astfel încât să primească și numele (unic) al agentului aferent. În continuare este dat un exemplu de constructor al interfeței grafice aferente unui agent JADE:

```
public myAgent_interface(String agentName, myTestAgent localAgent) {
```

agentName este folosit la identificarea GUI și localAgent este folosit la comunicația cu agentul aferent.

Capitolul 8: Realizarea unei platforme tolerante la defect

Cuprins

1. Toleranța la defect în JADE

În cadrul sistemelor multi-agent considerate, toleranța la defecte are două aspecte: în primul rând platforma trebuie să supraviețuiască unui defect (agentul AMS care reprezintă platforma trebuie replicat) și în al doilea rând serviciile individuale oferite de agenți trebuie să își continue operarea și după oprirea unui anumit agent. Cele două elemente se numesc toleranță la defect la nivel de platformă și toleranță la defect la nivel de agent (serviciu). În cele ce urmează (secțiune 2, respectiv secțiune 4) sunt detaliate modalitățile de realizare a unei platforme tolerante la defect, respectiv a unui agent care oferă un serviciu (implementează un protocol de interacțiune) tolerant la defect. Defectele abordate sunt: defecțiune hardware la nivel de platformă (aceasta generând o întrerupere a platformei în cazul cel mai defavorabil sau a unui agent singular în cazul cel mai favorabil), defecțiune la nivel de rețea (în acest caz, agenții separați de AMS se întrerup), respectiv defecțiune la nivel de aplicație (din rațiuni de programare un agent își întrerupe execuția).

În implementarea de aplicații industriale toleranța la defect este adesea una dintre cerințele principale care trebuie îndeplinite. JADE este o platformă distribuită, dar se bazează totuși pe un container principal pentru a găzdui servicii cheie, cum ar fi AMS și DF. Acesta este în mod clar un potențial punct unic de eșec (*single point of failure* – SPOF) care trebuie gestionat eficient pentru a se asigura că platforma rămâne pe deplin operațională chiar și în cazul unei defecțiuni a containerului principal. Acest lucru se realizează prin combinarea următoarelor două caracteristici:

- **replicarea containerului principal**. Această caracteristică, prezentată în secțiunea a 2-a a acestui capitol, permite replicarea containerului principal și a agentului AMS din interiorul acestuia, pentru a păstra toate replicile complet sincronizate și pentru a se asigura că, în caz de eșec, altul poate prelua;
- **persistența DF**. Această caracteristică, prezentată în secțiunea a 4-a a acestui capitol, permite înregistrarea catalogului agentului DF într-o bază de date relațională (DB). În cazul eșecului containerului principal, un nou agent DF este pornit automat pe noul container principal și își poate recupera catalogul din DB.

În teorie, există două abordări cunoscute (Fernández-Díaz, 2015; Fedoruk, 2002; Klein, 2003) pentru implementarea unui sistem software cu toleranță la defecte, cu disponibilitate ridicată: prima, intitulată abordare de supraviețuire (survivalist), păstrează funcționalitatea prin replicarea agenților, în timp ce a doua, intitulată abordare cetățenească (*citizen*), se bazează pe alți agenți care monitorizează execuția și iau măsuri corective în caz de defect. Principala diferență dintre cele două abordări constă în faptul că, în primul caz, agenții sunt proiectați să-și monitorizeze propria stare și să gestioneze excepții, în timp ce în al doilea caz entitățile externe monitorizează și supraveghează componenta tolerantă la defecțiuni. Mai mult, prima abordare bazată pe tehnici de replicare are două fațete: replicarea activă și pasivă. În cazul de replicare activă, toate replicile existente ale agentului efectuează aceleași operațiuni, au date sincronizate, iar ieșirea este generată de o singură entitate. Abordarea de tip supraviețuitor, bazată pe replicarea pasivă, menține doar o singură entitate activă și cele de rezervă sunt activate numai în caz de defecțiune; nu se dau informații despre starea internă a agentului care va eșua, cu excepția faptului că serviciul oferit continuă să fie furnizat.

Platforma JADE utilizează abordarea de replicare de tip supraviețuitor pentru a menține funcționalitatea atât a platformei, cât și a agenților componenți. Agenții din JADE sunt

structurați și localizați în containere care sunt procese Java interconectate și pot fi distribuite pe o rețea. În timpul inițializării platformei este creat un container principal care conține doi agenți critici pentru sistemul multi-agent: *Agent Management System* (AMS) și *Directory Facilitator* (DF). Chiar dacă este posibil să avem mai multe platforme și să trimitem mesaje între ele, toate celelalte containere trebuie să se alăture containerului principal pentru a avea o singură platformă multi-agent. Dacă containerul principal se defectează, AMS (care reprezintă MAS) și DF se închid și, în consecință, platforma MAS se oprește cu toate containerele sale conectate. Pentru a evita acest scenariu, pot fi lansate mai multe containere principale de rezervă care monitorizează activ masterul (containerul principal inițial). Acestea sunt grupate într-un inel de containere principale și pot utiliza două tipuri de politici pentru monitorizarea containerului principal: dinamic (serviciu de notificare adresă) și static (o listă fixă, cunoscută în prealabil, a containerelor principale). Containere simple pot fi adăugate la inelul containerelor principale și de rezervă, iar în aceste containere se află agenți. Dacă containerul principal master își încetează execuția, agenții AMS și DF sunt mutați automat într-un alt container principal de rezervă, asigurând astfel supraviețuirea platformei.

Dar supraviețuirea platformei nu este suficientă; agenții individuali care oferă servicii altor agenți pe baza unei cereri trebuie să fie, de asemenea, toleranți la defect. În acest scop, JADE oferă un mecanism intitulat agenți replicați virtuali. După cum sugerează și numele, acesta este un agent virtual (o adresă are în spate un set de agenți fizici), iar platforma se ocupă de direcționarea cererilor către diferite replici ale unui agent. Mesajele pot fi trimise fie către toate replicile disponibile, fie către o singură replică. Agentul virtual este un nume, nu un agent; nu are o locație, în timp ce replicile rulează pe containere bine definite, de preferință pe diferite gazde (cu adrese IP-uri diferite) pentru a asigura supraviețuirea în caz de defect. Platforma are grijă și de sincronizarea dintre variabilele interne ale replicilor.

2. Realizarea unei platforme tolerante la defect

Arhitectura distribuită JADE se bazează pe un nod special, numit *Main Container*, pentru a coordona toate celelalte noduri și a menține împreună întreaga platformă. Containerul principal JADE poate fi replicat pentru a implementa o platformă tolerantă la defecte cu Serviciul principal de replicare a containerelor (*Main Container Replication Service* – MCRS) implementat de clasa jade.core.replication.MainReplicationService. MCRS poate lansa orice număr de containere logice principale pe o platformă. Numai unul poate fi master, în timp ce

ceilalți trebuie să servească drept copie de rezervă. Toate containerele principale active constituie un inel logic în cadrul căruia se pot monitoriza reciproc. Dacă unul eșuează, celelalte containere principale de rezervă detectează evenimentul și iau acțiunile de recuperare corespunzătoare. Containerele non-principale se pot alătura platformei prin oricare dintre aceste noduri principale de container active, deoarece toate replicile sunt păstrate coerente și sincronizate între ele prin notificare încrucișată.

Deși majoritatea operațiunilor JADE sunt descentralizate, există câteva caracteristici esențiale care sunt realizate numai de containerul principal:

- **gestiunea tabelului de containere** (setul tuturor nodurilor care compun platforma distribuită);
- **gestiunea tabelului de descriptori globali** (setul tuturor agenților găzduiți de platforma distribuită, împreună cu locația lor actuală);
- **gestiunea tabelului de protocoale de transport a mesajelor** (*Message Transport Protocol* – MTP) (setul tuturor punctelor finale MTP implementate, împreună cu locația lor de implementare);
- **gestiunea agentului AMS care reprezintă platforma în sine**;
- **găzduirea platformei Agent DF implicit**.

Dacă containerul principal se termină sau devine indisponibil pentru celelalte containere ale platformei, toate caracteristicile anterioare devin indisponibile, rezultatul fiind închiderea totală, sau parțială a sistemului multi-agent. Închiderea parțială se referă la situația în care o parte din platformă se separă de containerul principal.

Pentru a menține platforma JADE pe deplin operațională chiar și în cazul unei defecțiuni a containerului principal, a fost introdus suportul pentru replicarea containerului principal. Folosind acest suport, este posibil să se pornească orice număr de noduri principale ale containerului (un container principal de tip master care deține efectiv AMS și un număr de containere principale de rezervă), care se vor aranja într-un inel logic, astfel încât la un defect individual celelalte containere vor observa și vor acționa în consecință. Containerele obișnuite se vor putea conecta la platformă prin oricare dintre nodurile aferente containerelor de tip *Main Container*; diferitele copii vor evolua împreună utilizând notificări încrucișate. Așa cum arată următoarea figură, fără replicarea containerului principal, platforma JADE are o topologie de tip stea; activarea replicării containerului principal transformă topologia într-un inel de containere principale.

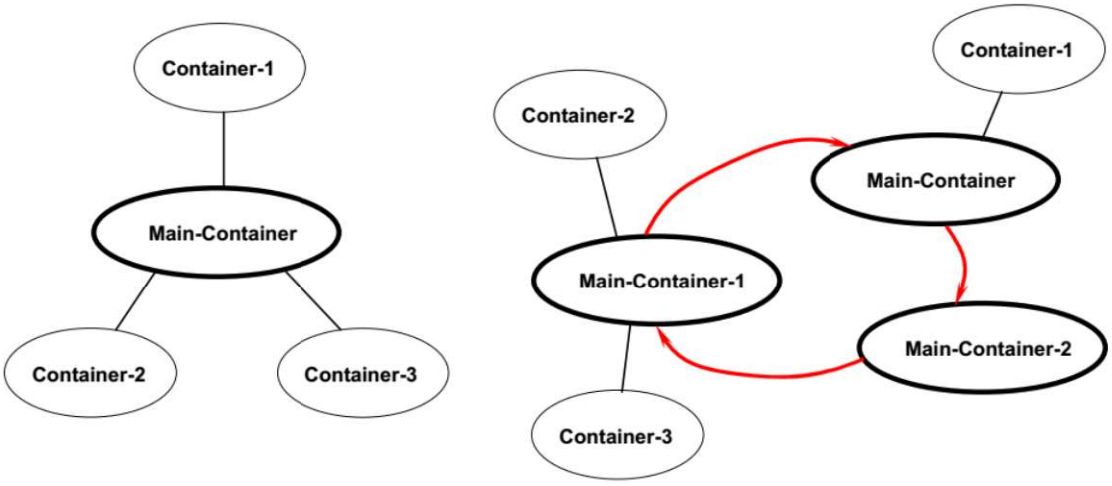

Fig.8.1 – Topologia unor platforme JADE cu (dreapta) și fără (stânga) toleranță la defect

În configurația tolerantă la defecte prezentată în partea dreaptă a figurii, trei noduri de tip container principal sunt aranjate într-un inel și fiecare nod își monitorizează vecinul: dacă nodul MainContainer-1 se defectează, nodul Main-Container-2 va observa și informa toate celelalte noduri principale ale containerului (în acest caz, doar containerul principal). Apoi, un inel mai mic va fi reconstruit (cu doar două elemente).

Figura arată, de asemenea, că elementele de tip container periferic pot fi distribuite în mod arbitrar între nodurile principale disponibile ale containerului. Orice container periferic este conectat la un singur nod și, în absența defecțiunilor, nu este complet conștient de toate celelalte copii ale sale. Când un nod de tip Main Container se defectează, vor exista în general niște containere periferice orfane (în exemplul curent, presupunând că un Main-Container-1 eșuează, Container-3 va deveni orfan). Când un container orfan detectează că nodul Main Container de care aparține nu mai este disponibil, acesta se atașează de un alt nod; pentru ca acest lucru să aibă succes, un container periferic trebuie să cunoască lista tuturor nodurilor principale ale containerului prezente pe platformă.

JADE acceptă două politici în distribuirea listei de noduri de tip Main Container către containerele periferice. O primă opțiune este activarea serviciului de notificare a adresei (*Address-Notification service*) pe toate nodurile containerului principal și pe containerele periferice. Acest serviciu va detecta adăugiri și eliminări la inelul nodurilor de container principal și va actualiza listele de adrese ale tuturor containerelor periferice implicate.

O a doua opțiune este de a trece lista de adrese la un container periferic de pornire cu argumentul liniei de comandă `-smaddrs`. Această abordare evită generarea traficului de notificare către containerele periferice, dar presupune o listă fixă de noduri de container principal, care este cunoscută în prealabil.

3. Exemplificare platformă tolerantă la defect

Lansarea unei platforme tolerante la defect din linia de comandă

O varietate de configurații de platformă tolerante la defect pot fi obținute prin diferite configurații din linia de comandă; în cele ce urmează sunt prezentate o serie de exemple, asociind o secvență de comenzi cu configurația platformei pe care o creează. În cele ce urmează, diferitele comenzi vor fi scrise la linia de comandă care cuprinde numele gazdei: `utilizator@gazda1`. Textul asociat liniei de comandă arată că numele de autentificare al utilizatorului a fost utilizat pentru conectarea la mașina `gazda1`. Comenzile date de utilizator vor fi prezentate în continuare.

Pas 1: Inițializarea a două noduri principale (tip Main Container) pe două mașini distincte

Pe mașina gazda1 se execută comanda următoare:

```
utilizator@gazda1: java jade.Boot -name Hydra -services
jade.core.replication.MainReplicationService;jade.core.replica
tion.AddressNotificationService
```

Comanda pornește un nod principal Master Container pe mașina gazda 1 și activează serviciile de replicare principală și notificare adresă pe acesta (de observat că o astfel de linie de comandă nu activează serviciile *Agent-Mobility* și *Notification*, necesare de exemplu utilizării agentului Sniffer). Opțiunea `-name` atribuie un nume („Hydra“ în acest caz) platformei constituite de containerul principal nou pornit.

```
utilizator@gazda2: java jade.Boot -backupmain -local-port 1234
-host host1 -port 1099 -services

jade.core.replication.MainReplicationService;jade.core.replica
tion.AddressNotificationServ ice
```

Comanda anterioară, dincolo de activarea celor două servicii necesare unei platforme tolerante la defecte, folosește opțiunea –backupmain pentru a specifica faptul că nodul nou pornit este un container principal, dar nu creează o nouă platformă separată, ci adaugă nodul ca element de rezervă în caz de defect. Mai degrabă, acest nou nod trebuie să se alăture unei platforme existente care este specificată de opțiunile -host și -port. În exemplul prezentat, aceste opțiuni indică exact Containerul principal care a fost pornit cu prima linie de comandă. Mai mult, opțiunea `-local-port` specifică portul pe care containerele noi trebuie să îl indice pentru a se conecta la acest container principal.

Reprezentarea grafică a platformei JADE rezultante este prezentată în figura 8.2.

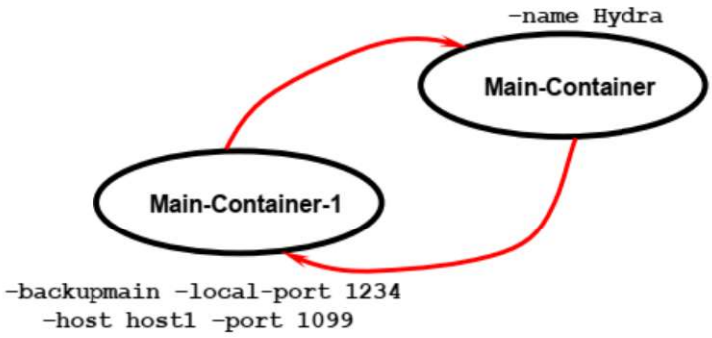

Fig.8.2 – Crearea unei platforme JADE tolerante la defect cu două containere principale

Pas 2: Adăugarea de containere secundare

Indiferent de ordinea lor de activare, cele două noduri principale ale containerului realizate în pasul anterior sunt absolut interschimbabile. Platforma construită anterior este o platformă unică, redundantă numită Hydra, expunând două adrese de gestiune (*Service Manager addresses*) (și anume, gazda1 cu portul 1099 și gazda2 cu portul 1234), astfel încât containerele periferice să se alăture. În cele ce urmează se va exemplifica demararea a două astfel de containere secundare (containere normale, fără opțiunea de backup), unul pentru fiecare adresă (și astfel unul pentru fiecare nod activ al containerului principal).

```
utilizator@gazda3: java jade.Boot -container -host gazda1 -port
1099 -services

jade.core.replication.AddressNotificationService
```

Comanda precedentă utilizează opțiunea -container pentru a lansa un container periferic și activează serviciul de notificare adresă pe acesta. Containerul se va alătura platformei Hydra folosind gazda cu numele `gazdă1` și portul 1099, conectându-se astfel la nodul principal (*Main Container*).

```
utilizator@gazda4: java jade.Boot -container -host gazda2 -port
1234
```

Această a doua comandă este similară cu prima, dar aici nu se activează serviciul AddressNotification; de fapt de la versiunea 3.6, acest serviciu nu mai este obligatoriu pentru ca toleranța la erori să funcționeze corect, el fiind necesar doar în cazul în care se dorește a se folosi agentul Sniffer cu toate funcționalitățile lui. Din moment ce se dorește ca acest container să se conecteze la nodul `Main-Container-1`, folosim opțiuni adecvate `-host` și `-port`.

Structura platformei JADE (conexiunile între containere) după acest pas este prezentată în figura următoare (Fig.8.3).

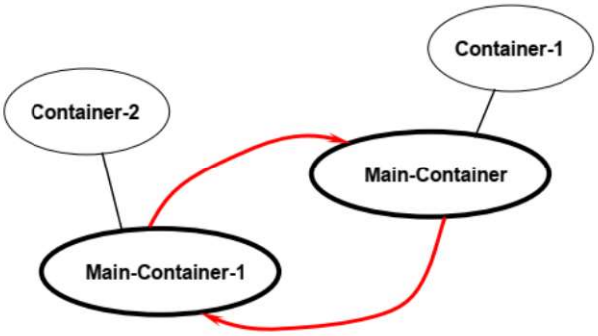

Fig.8.3 – Crearea unei platforme JADE tolerante la defect cu două containere principale și două containere secundare asociate celor două containere principale

Pas 3: Completarea platformei cu containere secundare și agenți

În acest pas final, se adaugă încă un nod principal și un nod secundar/periferic, astfel încât să se ajungă la o topologie de platformă tolerantă la defect, precum cea prezentată în figura 8.1.

```
utilizator@gazda5: java jade.Boot -backupmain -local-port 1099
-host gazda1 -port 1099 -services

jade.core.replication.MainReplicationService;jade.core.replica
tion.AddressNotificationService
```

Această comandă pornește un al treilea nod principal, exportând o adresă Service Manager pe gazda gazda5, portul 1099.

```
utilizator@gazda6: java jade.Boot -container -host gazda2 -port
1234
```

Comanda anterioară pornește un container periferic care se alătură platformei prin nodul MainContainer-1. Cu această ultimă comandă se obține topologia completă a platformei menționate anterior.

Toleranța la defect în cazul defectării unui nod principal

În acestă secțiune va fi descris cum o platformă tolerantă la defect se poate recupera și reconfigura singură după o eroare a containerului principal, continuând operațiunile normale. Pentru a exemplifica acest lucru, se va presupune că toți agenții software sunt găzduiți de cele trei containere periferice și că unul dintre nodurile de tip container principal se va termina brusc

ca urmare a unei defecțiuni (*Observație*: CTRL+F4 este diferit de CTRL+X, prima opțiune simulând defectul, în timp ce a doua opțiune închide platforma). Nodul afectat de eroare va fi nodul principal-container-1.

Procesul de recuperare poate fi observat dacă un agent RMA este pornit pe unul dintre containerele periferice neafectate de eroare. De îndată ce nodul Main-Container-1 se termină, nodul Main-Container-2 va observa și va informa nodul Main-Container. Va rezulta un inel nou, mai mic, de containere principale (compus din cele două noduri care au supraviețuit defectului). Nimic nu i se va întâmpla nodului Container-1, în timp ce celelalte două containere periferice se vor reconecta la un alt nod Principal Container imediat ce au nevoie de el. După acest defect simulat, noua topologie a platformei ar putea arăta precum cea descrisă în figura 8.4.

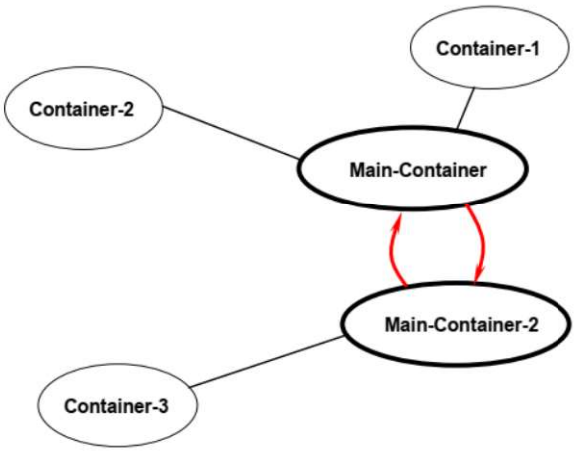

Fig.8.4 – Refacerea platformei ca urmare a unui defect suferit de un nod principal

Toate nodurile principale ale containerului sunt replici, cu o singură diferență. Doar una dintre replici (containerul principal master) „apare“ să găzduiască platforma AMS și agenții DF impliciți: aceasta este prima replică inițiată din inel, care este nodul principal-container în cazul de față. Defectarea anterioară a nodului Main-Container-1 a lăsat neatins AMS și DF implicit. În cazul defectării nodului Main-Container situația stă altfel. Evoluția evenimentelor este aceeași, nodul Main-Container-2 detectând defectul și cele trei containere periferice reconectându-se la acesta atunci când este necesar. De data aceasta, nodul MainContainer-2 își dă seama că nodul defect găzduia agenții de bază ai platformei (AMS și DF) și continuă să activeze propriile replici ale AMS și DF (fiecare nod principal al containerului are o replică latentă a celor două, care este actualizată la fel ca celelalte structuri de date la nivel de platformă). Din perspectiva unui agent RMA care rulează pe nodul Container-3, singura diferență este că Main-Container a dispărut de pe platformă și agenții AMS și DF s-au „mutat“ acum pe Main-Container-2.

4. Realizarea unui serviciu (agent) tolerant la defect – agenți virtuali replicați

Un „agent replicat virtual“ (*VR agent*) este o entitate virtuală: este doar un nume (mai precis un AID) care nu are asociat un agent de tip software ce se execută pe un sistem de operare . Pe de altă parte, „în spatele“ acestui nume există unul sau mai mulți agenți „fizici“ numiți *replici ale agentului virtual*. După cum este descris în figura 8.5, JADE se ocupă de trimiterea mesajelor destinate agentului virtual către o replică adecvată. Trebuie remarcat faptul că fiecare agent replică are propriul său nume, care diferă de cel al agentului virtual. Un agent virtual nu are o locație, în timp ce fiecare agent replică rulează pe un container/gazdă bine definit.

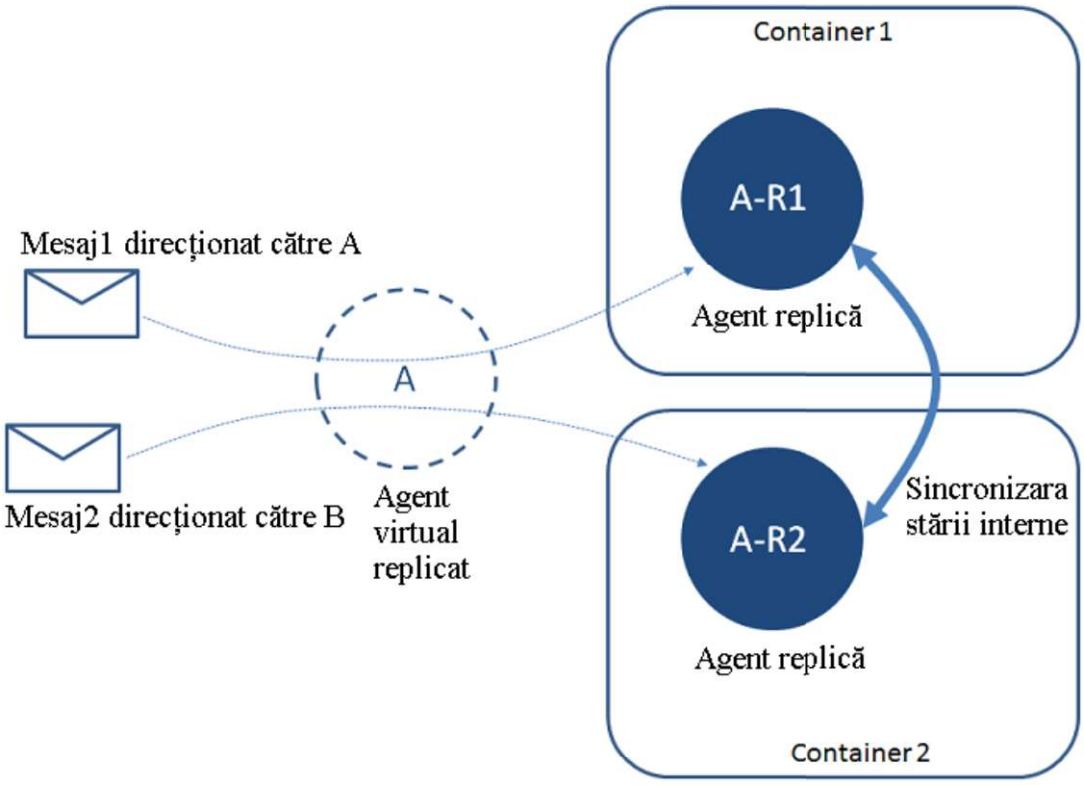

Fig.8.5 – Agent virtual replicat (*Virtual Replicated Agent*)

Este furnizat un mecanism simplu, dar flexibil, pentru a menține sincronizată starea internă a fiecărei replici a unui agent virtual. Prin urmare, agenții VR sunt un mijloc puternic de a atinge scalabilitatea și toleranța la defect. De exemplu, cu referire la figura 8.5, toate cererile trimise agentului virtual A sunt trimise automat către replicile atât A-R1 cât și A-R2. În cazul în care Container-1, unde se execută replica A-R1, se blochează brusc, alți agenți care interacționează cu agentul virtual A nici măcar nu își dau seama de acest lucru, deoarece toate mesajele vor fi trimise automat de JADE doar pentru a replica A-R2.

Prima replică a unui agent virtual se numește replică master. Celelalte replici pot fi create în orice moment și starea lor internă este copiată din cea a replicii master. Mai mult, replica principală este notificată despre adăugarea și eliminarea altor replici.

Agenții virtuali necesită serviciile *Agent Replication* (clasa `jade.core.replication.AgentReplicationService`) și *Agent Mobility* în toate containerele platformei. Primul oferă funcții de replicare, în timp ce al doilea este utilizat pentru a clona replica master atunci când trebuie creată o nouă replică.

5. Exemplificare agenți virtuali replicați

Față de situația descrisă anterior, în care este configurată doar platforma pentru a fi tolerantă la defecte, cazul unui serviciu (oferit de un agent software) tolerant la defect este mai complex. În primul rând, acesta trebuie să opereze pe o platformă tolerantă la defecte deoarece, în cazul cel mai defavorabil, poate pica chiar un nod principal. Astfel, în cele ce urmează este prezentat un exemplu simplu cu un agent care furnizează o valoare. Acest agent va fi replicat folosind soluția de agent virtual replicat și în același timp se va exemplifica actualizarea stării acestuia (reprezentată de valoare furnizată).

Codul proiectului care conține agentul furnizor de valoare, care este inclus în pachetul `examples.replication` al distribuției complete JADE, precum și codul agentului care interoghează agentul virtual vor fi prezentate în secțiunea următoare.

Agentul virtual replicat care oferă o valoare

Acest agent are o stare internă minimalistă, reprezentată de o valoare numerică. O interfață grafică simplă (Fig.8.6) permite utilizatorului să efectueze două lucruri: a) să aleagă o valoare numerică ce reprezintă starea internă și b) să adăuge clone suplimentare pentru susținerea serviciului agentului virtual replicat.

Este posibil să se solicite agentului `ValueProviderAgent` să furnizeze valoarea sa numerică internă prin intermediul acțiunii `GetValue` a `ValueManagementOntology`. Acesta este agentul care arată de fapt utilizarea agenților virtuali replicați.

Modul de lucru este următorul: se definește ca o replică principală (master) a unui agent replicat virtual și include o soluție pentru a crea noi replici. Doar replica master își păstrează interfața grafică (GUI).

Agentul care interoghează serviciul tolerant la defect (ValueReaderAgent)

Acest agent solicită periodic agentului virtual replicat să furnizeze valoarea sa numerică internă și apoi o afișează la consolă. Scopul său este doar de a arăta că, indiferent de replica utilizată, valoarea returnată este de fapt cea afișată în interfața grafică a agentului principal. Mai mult, replicile (inclusiv masterul) pot fi terminate și apoi create în mod dinamic fără a afecta deloc răspunsurile primite de agentul care interoghează serviciul (`ValueReaderAgent`).

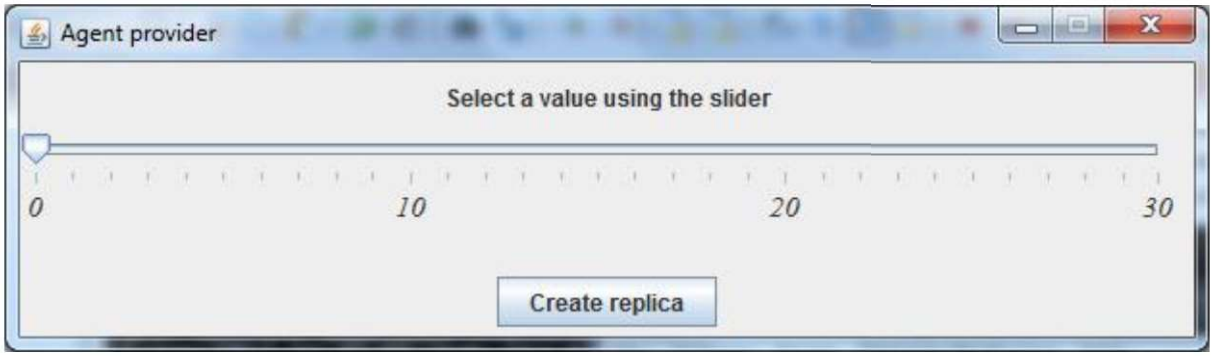

Fig.8.6 – Interfața grafică a agentului care oferă serviciul

Pentru a încerca exemplul Agentului furnizor de valoare, trebuie urmați pașii descriși în continuare. Se descarcă framework-ul JADE în formula completă constând în fișierele binare/compilate, surse, documentație și exemple. În figura 8.7 este prezentată locația codului aferent agentului virtual replicat, a agentului care oferă un serviciu și a agentului care consumă serviciul.

```
...
 |--jade/
      |--...
      |--lib/
          |--jade.jar
          |--jadeExamples.jar
          |--...
      |
      |--src/
          |--...
          |--examples/
                |--...
                |--replication/
                |-- sources of the Value Provider Agent example
```

Fig.8.7 – Structura arborescentă a frameworkului complet
și localizarea exemplului cu agenții virtuali replicați

Se deschide o consolă și apoi se specifică locația căii curente în directorul JADE, pentru ca apoi să se lanseze containerul principal cu agenții `ValueProviderAgent` și ValueReaderAgent atașați prin tastarea comenzii:

```
java -cp lib/jade.jar;lib/jadeExamples.jar jade.Boot -gui -
services jade.core.event.NotificationService;
jade.core.mobility.AgentMobilityService;
jade.core.replication.AgentReplicationService -agents
provider:examples.replication.ValueProviderAgent;reader:exampl
es.replication.ValueReaderAgent
```

Se acționează bara de selecție din interfața grafică aferentă agentului `ValueProviderAgent` și se verifică dacă `ValueReaderAgent` tipărește valoarea așteptată pe ieșirea standard. Ulterior se mai adaugă un agent de tip oferire serviciu, prin deschiderea unei console noi din directorul JADE și lansarea unui container periferic tastând:

```
java -cp lib/jade.jar;lib/jadeExamples.jar jade.Boot -
container -services
jade.core.event.NotificationService;jade.core.mobility.AgentMo
bilityService;jade.core.replication.AgentReplicationService
```

Acum, prin intermediul butonului de creare a unei replici (*Create Replica*) din interfața grafică a ValueProviderAgent se creează o nouă replică în containerul periferic nou creat.

Se modifică din nou valoarea numerică internă mutând elementul grafic de glisare și în același timp se urmăresc mesajele tipărite pe ieșirea standard pentru a se observa cum se modifică valoarea oferită.

Și valoarea numerică internă a replicii nou create este actualizată.

Solicitările pentru obținerea valorii numerice trimise de ValueReaderAgent sunt distribuite automat pe ambele replici.

În cele din urmă, este întreruptă abrupt (defecțiune) replica master prin intermediul interfeței grafice de administrare a JADE. Interfața grafică a agentului furnizor de serviciu dispare și reapare imediat. Acest lucru se datorează faptului că replica ce rulează pe Container-1 devine noua replică master și reacționează la acest eveniment, făcându-i vizibilă interfața grafică.

Mai mult, solicitările trimise de ValueReaderAgent sunt complet direcționate către replica de pe Container-1, astfel încât pentru ValueReaderAgent este transparent cine îi furnizează valoarea chiar dacă vechea replică master a dispărut brusc.

6. Considerații asupra codului aferent agentului virtual replicat

Pentru vizualizarea codului complet aferent agentului virtual replicat, se recomandă accesarea directorului cu exemple. Pentru a facilita înțelegerea aspectelor particulare ale unui agent virtual replicat, în cele ce urmează vor fi prezentate o serie de capturi de ecran cu liniile de cod etichetate.

a. Definirea agentului virtual replicat

Codul din figurile următoare prezintă metodele `setup()` și `takeDown()` ale agentului virtual replicat.

Analiza liniilor 10 și 11 arată că prima instrucțiune preia `ServiceHelper` al `AgentReplicationService`. A doua permite declararea faptului că acest agent este replica principală a unui agent replicat virtual. Parametrul HOT_REPLICATION specifică faptul că mesajele direcționate către agentul VR vor fi trimise în mod implicit la toate replicile disponibile. Dacă se utilizează COLD_REPLICATION, toate mesajele vor fi fost trimise numai la replica principală (alte replici utilizate numai în scopul toleranței la defect).

Metoda `makeVirtual()` returnează AID-ul agentului virtual replicat. Acesta este identificatorul agent (AID) de publicat în înregistrările DF (conform liniilor 13 – 23) pentru a exploata mecanismul agentului replicat virtual.

Liniile 24 – 46 înregistrează limbajele și ontologiile necesare, activează comportamentul care servește cererile pentru a obține valoarea internă și afișează interfața grafică aferentă agentului virtual replicat.

Liniile 48 – 54 conțin metoda takeDown () în cazul în care interfața grafică (dacă există) este închisă.

```
1    public class ValueProviderAgent extends Agent implements AgentReplicationHelper.Listener {
2
3         private transient ValueProviderAgentGui myGui;
4        private int myValue = 0;
5
6        @Override
7        protected void setup() {
8            try {
9                // Makes this agent become the master replica of a newly defined replicated agent
10               AgentReplicationHelper helper = (AgentReplicationHelper)
getHelper(AgentReplicationHelper.SERVICE_NAME);
11               AID virtualAid = helper.makeVirtual(getLocalName()+"_V",
AgentReplicationHelper.HOT_REPLICATION);
12
13               // Register to the DF.
14               // NOTE: we use the virtual agent AID (not the concrete agent AID).
15               // In this way requests from remote agents will be automatically spread across
16               // all replicas to achieve load balancing and fault tolerance.
17               DFAgentDescription dfad = new DFAgentDescription();
18               dfad.setName(virtualAid);
19               ServiceDescription sd = new ServiceDescription();
20               sd.setType("ValueProvider");
21               sd.setName("VirtualValueProvider");
22               dfad.addServices(sd);
23               DFService.register(this, dfad);
24
25               // Register required ontologies and language codecs
26               getContentManager().registerLanguage(new SLCodec());
27               getContentManager().registerOntology(ValueManagementOntology.getInstance());
28
29               // Add the behaviour serving requests to read our current value
30               addBehaviour(new OntologyServer(this, ValueManagementOntology.getInstance(),
ACLMessage.REQUEST, this));
31
32               // Show the GUI that allows the user to set the value and to create other replicas
33               myGui = new ValueProviderAgentGui(this, myValue);
34               myGui.setVisible(true);
35           }
36           catch (ServiceException se) {
37               System.out.println("Agent "+getLocalName()+" - Error retrieving AgentReplicationHelper!!!
Check that the AgentReplicationService is correctly installed in this container");
38               se.printStackTrace();
39               doDelete();
40           }
41           catch (FIPAException fe) {
42               System.out.println("Agent "+getLocalName()+" - Error registering with the DF");
43               fe.printStackTrace();
44               doDelete();
45           }
46       }
47
48       @Override
49       protected void takeDown() {
50           // Close the GUI (if present) when the agent terminates
51           if (myGui != null) {
52               myGui.dispose();
53           }
54       }
```

Fig.8.8 – Codul aferent metodelor `setup()` și `takeDown()` ale agentului virtual replicat

b. Crearea de replici noi

Următorul fragment de cod arată metoda care este apelată atunci când utilizatorul selectează butonul de creare replică din interfața grafică a agentului ValueProvider și selectează un nume și o locație pentru noua replică.

```
1       void createReplica(String replicaName, String where) {
2           if (replicaName == null || replicaName.trim().length() == 0) {
3               System.out.println("Replica name not specified");
4               return;
5           }
6           if (where == null || where.trim().length() == 0) {
7               System.out.println("Replica location not specified");
8               return;
9           }
10          try {
11              AgentReplicationHelper helper = (AgentReplicationHelper)
getHelper(AgentReplicationHelper.SERVICE_NAME);
12              helper.createReplica(replicaName.trim(), new ContainerID(where.trim(), null));
13          }
14          catch (Exception e) {
15              System.out.println("Agent "+getLocalName()+" - Error creating replica on container "+where);
16              e.printStackTrace();
17          }
18      }
```

Fig.8.9 – Codul aferent creării unui nou agent de tip replicat

Liniile de la 2 la 9 efectuează doar verificări ale numelui și locației inserate de utilizator. Metoda `createReplica()` a AgentReplicationHelper face clonarea efectivă. După cum s-a menționat, crearea reală a replicii se face prin clonarea replicii master. Astfel, agentul trebuie să fie complet serializabil și metoda `afterClone()` trebuie redefinită pentru a reinițializa câmpuri tranzitorii, cum ar fi limbajele de conținut înregistrate și ontologiile.

Următorul fragment de cod arată utilizarea metodei `afterClone()`.

```
1       @Override
2       public void afterClone() {
3           // New replicas are created cloning the master replica.
4           // Just after cloning restore transient field such as registered ontologies and language codecs
5           System.out.println("Agent "+getLocalName()+" - Alive");
6           getContentManager().registerLanguage(new SLCodec());
7           getContentManager().registerOntology(ValueManagementOntology.getInstance());
8       }
```

Fig.8.10 – Codul aferent metodei procesului ulterior clonării

c. Sincronizarea tuturor replicilor

Următorul fragment de cod arată metoda care este apelată de interfața grafică ori de câte ori se setează valoarea numerică internă prin intermediul interfeței grafice. În acest scenariu minimal, o astfel de valoare, memorată în câmpul `myValue` al clasei ValueProviderAgent, reprezintă pe deplin starea internă a agentului.

```
1       public void setValue(int newValue) {
2           // The call to setValue() will be invoked on other replicas too
3           AgentReplicationHandle.replicate(this, "setValue", new Object[]{newValue});
4
5           myValue = newValue;
6           System.out.println("Agent "+getLocalName()+": VALUE = "+myValue);
7       }
```

Fig.8.11 – Codul aferent modificării stării interne a unui agent

Linia 3 asigură în special că metoda setValue() este apelată la toate celelalte replici. Prin urmare, sincronizarea tuturor replicilor unui agent virtual replicat se realizează prin replicarea apelurilor către metode care se așteaptă să modifice starea internă a agentului. Este responsabilitatea programatorului să definească ce apel să reproducă și cum.

Trebuie avut în vedere că metoda replicate() nu este apelată de `AgentReplicationHelper`, așa cum s-a întâmplat pentru alte caracteristici ale mecanismului agentului replicat virtual, ci este furnizată ca metodă statică a unei clase de tip `AgentReplicationHandle`. Acest lucru se face pentru a permite apelarea transparentă a acestei metode chiar și atunci când AgentReplicationService nu este instalat (desigur, în acest caz, metoda nu are niciun efect). În acest fel, este posibil să se proiecteze un agent pentru a exploata mecanismul agentului replicat virtual, dar pentru a activa replicarea numai atunci când este efectiv necesar.

d. Gestiunea evenimentelor asociate replicilor

Privind linia 1 din primul fragment de cod prezentat în secțiunea 6.a, trebuie reținut faptul că interfața `AgentReplicationHelper.Listener` este implementată cu clasa `ValueProviderAgent`. Aceasta spune serviciului de bază `AgentReplication` că replica master trebuie să fie notificată ori de câte ori se întâmplă un eveniment legat de replică. Acest lucru se face invocând metodele interfeței `AgentReplicationHelper.Listener`.

- `replicaAdded()` – notifică replica principală că a fost creată cu succes o nouă replică. Procesul de creare a copiei are loc asincron;
- `replicaCreationFailed()` – notifică replica master că a eșuat crearea unei noi replici;
- `replicaRemoved()` – notifică replica master că o replică existentă s-a defectat (oprire abruptă);

- `becomeMaster()` – notifică noua replică master selectată că replica master anterioară s-a defectat (oprire abruptă).

Fragmentul de cod următor arată implementarea metodei becomeMaster() în care noua replică master selectată își activează interfața grafică.

```
1       @Override
2       public void becomeMaster() {
3           // The old master replica is dead. I'm the new master replica --> Show the GUI
4           System.out.println("Agent "+getLocalName()+" - I'm the new master replica");
5           myGui = new ValueProviderAgentGui(this, myValue);
6           myGui.setVisible(true);
7       }
```

Fig.8.12 – Codul aferent metodei apelate la transformarea unui agent în agent principal

Referințe

- Active Components JADEX, https://www.activecomponents.org/#/docs/overview, consultatã în noiembrie 2022

- AnyLogic Simulation Software, https://www.anylogic.com/, consultatã în noiembrie 2022

- Barata, J. & Camarinha-Matos, Luis. (2003). Coalitions of manufacturing components for shop floor agility - the CoBASA architecture. IJNVO. 2. 50-77. 10.1504/IJNVO.2003.003518.

- Bellifemine, F., Carie, G., Greenwood, D., 2007, Developing multi-agent systems with JADE, Wiley, ISBN 978-0-470-05747-6

- Bergenti, Federico & Caire, Giovanni & Gotta, Danilo. (2014). Agent-based social gaming with AMUSE. Procedia Computer Science. 32. 914-919. 10.1016/j.procs.2014.05.511.

- Bhamra, Dr. G. & Patel, Ram & Verma, Anil. (2014). Intelligent Software Agent Technology: An Overview[J]. International Journal of Computer Applications. 89. 19-31. 10.5120/15474-4160

- Bond, A. H., Gasser, L., editors. Readings in Distributed Artificial Intelligence. Morgan Kaufmann Publishers: San Mateo, CA, 1988

- Borangiu, T., Răileanu, S., Anton, F., Pârlea, M., Tahon, C., Berger, T., Trentesaux, D. (2011). Product-driven automation in a service oriented manufacturing cell, International Conference on Industrial Engineering and Systems Management

- Braubach, Lars & Pokahr, Alexander. (2006). Jadex: A BDI-Agent System Combining Middleware and Reasoning. 10.1007/3-7643-7348-2_7.

- Brian Yueshuai He, Jinkai Zhou, Ziyi Ma, Ding Wang, Di Sha, Mina Lee, Joseph Y.J. Chow, Kaan Ozbay (2021), A validated multi-agent simulation test bed to evaluate congestion pricing policies on population segments by time-of-day in New York City, Transport Policy, Volume 101, Pages 145-161, ISSN 0967-070X

- Cardin, O., Trentesaux, D., Thomas, A., Castagna, P., Berger, T., Bril, H. (2015). Coupling Predictive Scheduling and Reactive Control in Manufacturing: State of the Art and Future Challenges. In: Borangiu, T., Thomas, A., Trentesaux, D. (eds) Service Orientation in Holonic and Multi-agent Manufacturing. Studies in Computational Intelligence, vol 594. Springer, Cham. https://doi.org/10.1007/978-3-319-15159-5_3

- Choi I. -S., Hong J., Kim T. -W. (2020), Multi-Agent Based Cyber Attack Detection and Mitigation for Distribution Automation System, IEEE Access, vol. 8, pp. 183495-183504, 2020, doi: 10.1109/ACCESS.2020.3029765.

- Costa-Montenegro, Enrique & Burguillo, Juan & Rodríguez Hernández, Pedro & Gonzalez-Castano, Francisco & Curras-Parada, Maria & Gomez-Rana, Patricia & Rey-Souto, Juan. (2008). Multi-Agent System Model of a BitTorrent Network. 586 - 591. 10.1109/SNPD.2008.168.

- Derigent, W., Cardin, O. & Trentesaux, D. (2021), Industry 4.0: contributions of holonic manufacturing control architectures and future challenges. J Intell Manuf 32, 1797–1818. https://doi.org/10.1007/s10845-020-01532-x

- Dmitri Muravev, Hao Hu, Aleksandr Rakhmangulov, Pavel Mishkurov (2021), Multi-agent optimization of the intermodal terminal main parameters by using AnyLogic simulation platform: Case study on the Ningbo-Zhoushan Port, International Journal of Information Management, Volume 57, 102133, ISSN 0268-4012,

- Duangsuwan, Jarunee & Liu, Kecheng. (2010). A Multi-agent System for Intelligent Building Control - Norm Approach.. ICAART 2010 - 2nd International Conference on Agents and Artificial Intelligence, Proceedings. 2. 22-29.

- Eric Marcon, Sondes Chaabane, Yves Sallez, Thérese Bonte, Damien Trentesaux (2017), A multi-agent system based on reactive decision rules for solving the caregiver routing problem in home health care, Simulation Modelling Practice and Theory, Volume 74, Pages 134-151, ISSN 1569-190X,

- Ferber, J., Les systemes multi-agents – vers une intelligence collective (1995), InterEditions, Paris (ISBN 2-7296-0572-X)

- Franklin, S., Graesser, A., Is it an Agent, or just a Program? (1996) : A Taxonomy for Autonomous Agents, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag

- Fernández-Díaz A., Benac-Earle C., Fredlund L., Adding distribution and fault tolerance to Jason, Science of Computer Programming, Volume 98, Part 2, 1 February 2015, Pages 205-232

- Fedoruk A., Deters R., Improving fault-tolerance by replicating agents, in: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2, AAMAS'02, ACM, New York, NY, USA, ISBN 1-58113-480-0, 2002, pp. 737-744, http://doi.acm.org/10.1145/544862.544917

- George, Abraham & Ali, Mohammad & Papakostas, Nikolaos. (2021). Utilising robotic process automation technologies for streamlining the additive manufacturing design workflow. CIRP Annals. 70. 10.1016/j.cirp.2021.04.017.

- Ghid utilizator WindowBuilder, https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.wb.doc.user%2Fhtml%2Findex.html, consultatã în noiembrie 2022

- Gutknecht, O. & Ferber, J. (2000). Madkit: a Generic Multi-Agent Platform. Autonomous Agents. AGENTS 2000, Barcelona, ACM Press, 78–79

- Herrera, Manuel & Perez Hernandez, Marco & Parlikad, Ajith Kumar & Izquierdo, Joaquín. (2020). A Review on Control and Optimisation of Multi-Agent Systems and Complex Networks for Systems Engineering. 10.20944/preprints202001.0282.v1.

- Herrero, Álvaro & Corchado, Emilio. (2009). Multiagent Systems for Network Intrusion Detection: A Review. Advances in Intelligent and Soft Computing. 63. 143-154. 10.1007/978-3-642-04091-7_18.

- http://jade.tilab.com/doc/tutorials/JADEAdmin/index.html, consultatã în noiembrie 2022

- http://www.aosgrp.com.au/, paginã oficialã JACK, consultatã în noiembrie 2022

- http://www.erlang.org/, paginã oficialã Erlang, consultatã în noiembrie 2022

- http://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html, consultatã în noiembrie 2022

- https://jade.tilab.com/, paginã oficialã JADE, consultatã în noiembrie 2022

- IEEE Recommended Practice for Industrial Agents: Integration of Software Agents and Low-Level Automation Functions, Developed by the Standards Committee of the IEEE Industrial Electronics Society, Approved 24 September 2020, IEEE SA Standards Board

- JACK autonomous software, https://aosgrp.com/products/jack/, accessed in June 2022

- James Odell, Agent Technology - An Overview (2011), paper/booklet, http://www.jamesodell.com/Agent_Technology-An_Overview.pdf, accessed in June 2022

- JAVA Agent DEvelopment Framework, https://jade.tilab.com/, consultatã în noiembrie 2022

- Jennings, N.R., Sycara, K., Wooldridge, M. (1998), A Roadmap of Agent Research and Development, Autonomous Agents and Multi-Agent Systems, 1, 7–38

- Klein M., Rodriguez-Aguilar J.-A., Dellarocas C., Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death, Auton. Agents Multi-Agent Syst. (ISSN 1387-2532) 7 (2003) 179-189, http://dx.doi.org/10.1023/A:1024145408578.

- Kravari, Kalliopi & Bassiliades, Nick. (2015). A Survey of Agent Platforms. Journal of Artificial Societies and Social Simulation. 18. 10.18564/jasss.2661.

- Kruger, Karel & Basson, Anton. (2019). Evaluation of JADE multi-agent system and Erlang holonic control implementations for a manufacturing cell. International Journal of Computer Integrated Manufacturing. 32. 1-16. 10.1080/0951192X.2019.1571231.

- Krzywicki, Daniel & Turek, W. & Byrski, Aleksander & Kisiel-Dorohinicki, Marek. (2015). Massively-concurrent Agent-based Evolutionary Computing. Journal of Computational Science. 11. 10.1016/j.jocs.2015.07.003.

- Lan Lu, Gong Wang (2008), A study on multi-agent supply chain framework based on network economy, Computers & Industrial Engineering, Volume 54, Issue 2, Pages 288-300, ISSN 0360-8352,

- Lane, Justin. (2014). Method, Theory, and Multi-Agent Artificial Intelligence: Creating computer models of complex social interaction. Journal for the Cognitive Science of Religion. 1. 161-180. 10.1558/jcsr.v1i2.161.

- Lars Braubach, Alexander Pokahr and Winfried Lamersdorf (2005), Jadex: A BDI-Agent System Combining Middleware and Reasoning, DOI:10.1007/3-7643-7348-2_7

- Latsou, Christina & Farsi, Maryam & Erkoyuncu, John & Morris, Geoffrey. (2021). Digital Twin Integration in Multi-Agent Cyber Physical Manufacturing Systems. IFAC-PapersOnLine. 54. 811-816. 10.1016/j.ifacol.2021.08.096

- Leitao, P., Colombo, A.W., Karnouskos, S., (2016), Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges, Computers in Industry, Volume 81, Pages 11-25, ISSN 0166-3615

- Leitao, P., Karnouskos, Stamatis & Ribeiro, Luis & Moutis, Panayiotis & Barbosa, José & Strasser, Thomas. (2017). Common practices for integrating industrial agents and low level automation functions. 6665-6670. 10.1109/IECON.2017.8217164.

- Leitao, P., Maŕik V., Vrba P. (2013). Past, Present, and Future of Industrial Agent Applications. Industrial Informatics, IEEE Transactions on. 9. 2360-2372. 10.1109/TII.2012.2222034.

- Leitao, P., Restivo, F. (2006), ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control, Computers in Industry, Vol. 57, No. 2, 121-130

- Leitao, P., S. Karnouskos, L. Ribeiro, J. Lee, T. Strasser, Colombo A. W. (2016), "Smart agents in industrial cyber–physical systems," Proceedings of the IEEE, vol. 104, no. 5, pp. 1086–1101.

- Leszczyna R. (2008), Evaluation of Agent Platforms (ver. 2.0). EUR 23508 EN. Luxembourg (Luxembourg): European Commission; JRC47224

- Leszczyna R. Evaluation of Agent Platforms (ver. 2.0) (2008). EUR 23508 EN. Luxembourg (Luxembourg): European Commission. JRC47224

- Leszczyna, R. (2006). Architecture Supporting Security of Agent Systems. PhD thesis, Gdansk University of Technology, Gdansk, Poland.

- Lyu, Guolin & Fazlirad, Alireza & Brennan, Robert. (2020). Multi-Agent Modeling of Cyber-Physical Systems for IEC 61499 Based Distributed Automation. Procedia Manufacturing. 51. 1200-1206. 10.1016/j.promfg.2020.10.168

- M. S. Simoiu, I. Fagarasan, S. Ploix, V. Calofir and S. S. Iliescu (2022), Towards Energy Communities: A Multi-Agent Case Study, IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1-6, doi: 10.1109/AQTR55203.2022.9802060.

- MaDKit, The Multiagent Development Kit, https://www.madkit.net/madkit/, consultatã în noiembrie 2022

- Marta Ginovart, Clara Prats (2012), A Bacterial Individual-Based Virtual Bioreactor to Test Handling Protocols in a Netlogo Platform, IFAC Proceedings Volumes, Volume 45, Issue 2, Pages 647-652, ISSN 1474-6670, ISBN 9783902823236

- Mcfarlane, Duncan & Giannikas, Vaggelis & Wong, Alex & Harrison, Mark. (2013). Product intelligence in industrial control: Theory and practice. Annual Reviews in Control. 37. 69-88. 10.1016/j.arcontrol.2013.03.003.

- Mesa: Agent-based modeling in Python 3+, https://mesa.readthedocs.io/en/latest/, consultatã în noiembrie 2022

- Meyer, Gerben & Främling, Kary & Holmström, Jan. (2009). Intelligent Products: A survey. Computers in Industry. 60. 137-148. 10.1016/j.compind.2008.12.005.

- Nancy Jamison (2017), Robotic Process Automation: A New Era of Agent Engagement, A Frost & Sullivan White Paper, www.frost.com

- NetLogo, https://ccl.northwestern.edu/netlogo/, accessed in June 2022

- North, MJ, NT Collier, J Ozik, E Tatara, M Altaweel, CM Macal, M Bragen, and P Sydelko (2013), Complex Adaptive Systems Modeling with Repast Simphony, Complex Adaptive Systems Modeling, Springer, Heidelberg, FRG. https://doi.org/10.1186/2194-3206-1-3.

- Obitko, M., Maøík, V., Ontologies for Multi-Agent Systems in Manufacturing Domain, DEXA Workshop, pp. 597-602, 2002

- Obitko, M., Maøík, V., Ontologies for Multi-Agent Systems in Manufacturing Domain, DEXA Workshop, pp. 597-602, 2002

- Pach, Cyrille & Berger, Thierry & Sallez, Yves & Trentesaux, Damien. (2012). Instantiation of the Open-Control Concept in FMS based on Potential Fields. IECON Proceedings (Industrial Electronics Conference). 10.1109/IECON.2012.6389486.

- Pechoucek, M., Marik, V. (2008), Industrial deployment of multi-agent technologies: review and selected case studies, Autonomous Agents and Multi-Agent Systems, 397–431

- Răileanu S., (2011), Proposition of a generic model for the control of a guided flow system / Application of the holonic concepts in intelligent transportation (FMS/PRT), PhD thesis.

- Răileanu, S., Borangiu, T., Radulescu, S. (2014). Towards an Ontology for Distributed Manufacturing Control. Studies in Computational Intelligence. 544. 97-109. 10.1007/978-3-319-04735-5-7.

- Roehrich, J.K.; Parry, G. and Graves, A. (2011), Implementing build-to-order strategies: enablers and barriers in the European automotive industry, International Journal of Automotive Technology and Management. 11(3): 221-235

- Romulus-Catalin Damaceanu (2008), An agent-based computational study of wealth distribution in function of resource growth interval using NetLogo, Applied Mathematics and Computation, Volume 201, Issues 1–2, Pages 371-377, ISSN 0096-3003

- Sallez, Yves & Berger, Thierry & Trentesaux, Damien. (2009). Management du cycle de vie d'un produit actif : Concept d'agent d'augmentation.

- Smith (1980). "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver". IEEE Transactions on Computers. C-29 (12): 1104–1113. doi:10.1109/TC.1980.1675516. ISSN 0018-9340. S2CID 15267324

- Souissi, Mohamed & Bensaid, Khalid & Rachid, Ellaia. (2018). Multi-agent modeling and simulation of a stock market. Investment Management and Financial Innovations. 15. 123-134. 10.21511/imfi.15(4).2018.10.

- Taherian, Mostafa & Mousavi, Seyed & Chamani, Hooman. (2018). An agent-based simulation with NetLogo platform to evaluate forward osmosis process (PRO Mode). Chinese Journal of Chemical Engineering. 26. 2487-2494. 10.1016/j.cjche.2018.01.032.

- The Foundation for Intelligent Agents, http://www.fipa.org/, consultatã în noiembrie 2022

- The Repast Suite, https://repast.github.io/, consultatã în noiembrie 2022

- Valckenaers, Paul & Brussel, H.. (2015). Design for the Unexpected, From Holonic Manufacturing Systems towards a Humane Mechatronics Society.

- Wilensky, U. & Rand, W. (2015). Introduction to Agent-Based Modeling: Modeling Natural, Social and Engineered Complex Systems with NetLogo. Cambridge, MA. MIT Press

- Winikoff, Michael. (2005). Jack Intelligent Agents: An Industrial Strength Platform, pp.175-193, in Multi-Agent Programming Languages, Platforms and Applications, Edited by Rafael H. Bordini, Jiirgen Dix, Mehdi Dastani, Amal El Fallah Seghrouchni

- Woltmann, Stefan & Kittel, Julia & Stomberg, Michael & Coordes, Andre. (2020). Using Multi-Agent Systems for Demand Response Aggregators: A Technical Implementation. 10.1109/ETFA46521.2020.9212168

- Wong, C., McFarlane, D., Zaharudin, A., & Agarwal, V. (2002). The intelligent product driven supply chain. 2002 IEEE international conference on systems, man and cybernetics (Vol. 4, pp. 6). IEEE.

- Wooldridge, Michael J. (2002). An Introduction to Multi-agent Systems. New York: John Wiley & Sons. p. 27. ISBN 0-471-49691-X

- Workflows and Agents Development Environment, https://jade.tilab.com/wadeproject/, consultatã în noiembrie 2022

- www.fipa.org, paginã oficialã Foundation for Intelligent and Physical Agents, consultatã în noiembrie 2022