Fundação Edson Queiroz
Universidade de Fortaleza – UNIFOR
Centro de Ciências Tecnológicas – CCT
Curso de Informática

Monografia de Conclusão de Curso

Estudo e Desenvolvimento de Sistemas Multiagentes usando JADE: *Java Agent Development framework*

por

Leonardo Ayres de Morais e Silva

Fortaleza – Ceará - Brasil

Junho de 2003

Fundação Edson Queiroz
Universidade de Fortaleza – UNIFOR
Centro de Ciências Tecnológicas – CCT
Curso de Informática

Estudo e Desenvolvimento de Sistemas Multiagentes usando JADE: *Java Agent Development framework*

Por

Leonardo Ayres de Morais e Silva

Monografia apresentada ao Centro de Ciências Tecnológicas da Universidade de Fortaleza como parte dos pré-requisitos para a obtenção da graduação de Bacharel em Informática

Área de Concentração : Inteligência Artificial
Orientador : Prof. João José Vasco Furtado, Dr.

Fortaleza – Ceará – Brasil

Junho de 2003

BANCA EXAMINADORA

_____

Pedro Porfírio Muniz Farias, Dr.

Examinador

_____

João José Vasco Furtado, Dr.

Orientador

APROVADA EM ___/___/___

Dedico este trabalho à minha família, amigos, professores e todas as outras pessoas que apoiaram e incentivaram mesmo nas dificuldades e que, de uma forma ou outra, me fizeram crescer.

AGRADECIMENTOS

Em primeiro lugar aos meus pais Rui e Maria, e às minhas irmãs, Flávia e Fernanda, pelo apoio e incentivo em todos os sentidos e que, mesmo com a distância, sempre estiveram ao meu lado oferecendo todas as condições possíveis para que eu conseguisse meus objetivos.

Aos meus amigos e colegas da Unifor, tanto da Informática como do Direito e de outros cursos, que compartilharam comigo estes anos de estudo e trabalho. Pelas experiências e amizades com os quais tive oportunidade de aprender muito. Em especial ao Vando, ao Bruno, ao Daniel Gonçalves e ao Daniel Cruz.

Ao meu orientador e professor Vasco por sua competência e por sua experiência.

Ao Eurico pela paciência e convivência e pela co-orientação neste trabalho.

A todos do Projeto ExpertCop e da Secretaria de Segurança Pública pelo conhecimento adquirido tanto no lado acadêmico quanto no lado pessoal.

A todos os professores da Unifor que, de uma forma ou de outra, contribuíram para o meu crescimento pessoal.

Aos meus amigos de Teresina, minha terra natal, os quais são muitos e por esse motivo seria impossível citar todos, pela amizade e pelas experiências de vida.

A todas as pessoas que me ajudaram direta e indiretamente para a realização de meus objetivos.

Por fim, a Deus.

Primeiro aprenda a ser um artesão. Isso não impedirá você de ser um Gênio. (Eugene Delacroix)

ABSTRACT

The concept of intelligent, autonomous and independent agents interacting each other, called Multi-agent System (MAS) has gradually received great attention of the academic and industrial area.

The recent advances in this research area has emerged divers MAS architectures aimed at facilitating the development of agent-based intelligent systems.

The objective of this work is to describe the technology of MAS, showing the potential of one framework for the development of agent applications called JADE. JADE (Java Agent DEvelopment framework) is an environment that possesses an infrastructure of support and development of a MAS, becoming its implementation robust and simple.

An application based on the problem of the Prey and the Hunter was developed using the JADE resources, aiming at to show all the potential that this environment considers to offer.

Keywords: Artificial Intelligence, Intelligent Agents, Multiagent Systems, JADE.

RESUMO

Sistemas multiagentes com seus conceitos de agentes inteligentes e autônomos interagindo juntos com um objetivo comum têm despertado grande interesse tanto na área acadêmica quanto na área industrial.

Os avanços recentes nessa área de pesquisa tem levado ao surgimento de muitas arquiteturas multiagentes e, com isso, uma necessidade de simplificação e padronização desses novos conceitos visando facilitar o desenvolvimento de sistemas inteligentes baseados em agentes.

O objetivo deste trabalho é abordar a tecnologia de agentes e de sistemas multiagentes, mostrando o potencial de um *framework* para o desenvolvimento de aplicações orientadas a agentes chamado JADE. JADE (*Java Agent DEvelopment framework)* é um ambiente que propõe toda uma infra-estrutura de suporte e desenvolvimento de sistemas multiagentes, tornando a implementação mais simples e robusta.

Para isso foi desenvolvida uma aplicação baseada no problema da Presa e do Caçador usando os recursos de JADE, visando mostrar todo o potencial que esse ambiente se propõe a oferecer.

Palavras-chave: Inteligência Artificial, Agentes Inteligentes, Sistemas Multiagentes, JADE.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

INTRODUÇÃO

Com a complexidade cada vez maior dos sistemas de computação, novos modelos de desenvolvimento vêm surgindo, dentre eles o modelo de desenvolvimento baseado em agentes.

A tecnologia de agentes tem adquirido nos últimos anos uma importância cada vez maior em muitos aspectos da computação, principalmente na área de Inteligência Artificial de Distribuída. O conceito de autonomia e de aplicações capazes de executar tarefas de forma inteligente e independente tem despertando grandes interesses tanto na área acadêmica. Interações entre agentes que trabalham juntos para um objetivo único e maior vem reforçando e amadurecendo o conceito de sistemas multiagentes.

Estudo e pesquisas sobre sistemas multiagentes dentro do campo da Inteligência Artificial Distribuída já começou cerca de 20 anos atrás. Atualmente esses sistemas não são mais apenas tópicos de pesquisa, mas já estão se tornando um importante assunto na área de ensino acadêmico e em aplicações comerciais e industriais. Como conseqüência disso, várias metodologias, arquiteturas e ferramentas já foram desenvolvidas para facilitar o desenvolvimento de sistemas multiagentes.

O objetivo principal deste trabalho é o foco em um ambiente de desenvolvimento de sistemas multiagentes chamado JADE: ***J****ava* ***A****gent* ***DE****velopment framework.* JADE simplifica o desenvolvimento fornecendo um *framework* completo que trata da comunicação, do ciclo de vida do agente, do monitoramento da execução, entre outras atividades. Muitos autores consideram JADE um *middleware,* ou seja uma camada intermediária de desenvolvimento, por oferecer um modelo que pode ser utilizado como base para implementação de modelos de mais alto nível.

JADE, como sua própria nomenclatura deixa claro, é totalmente baseado na linguagem Java. Ela possui muitos atributos e características que a tornam ideal na implementação de agentes e sistemas multiagentes.

Essa monografia está dividida em quatro capítulos. No primeiro capítulo abordaremos os conceitos de agentes e sistemas multiagentes, visando dar uma breve introdução sobre essas tecnologias. Falaremos também de FIPA (*Foundation For Intelligent Physical Agents)* que é uma entidade que vêm especificando padrões para o desenvolvimento de sistemas multiagentes. No capítulo dois falaremos do JADE propriamente dito, abordando sua arquitetura e funcionamento, seu relacionamento com a FIPA, interoperabilidade, entre outras características. No capítulo três falaremos da parte técnica do JADE: instalação, requisitos mínimos, a estruturação de um programa feito no ambiente e as ferramentas que JADE oferece ao desenvolvedor. No capítulo quatro falaremos de uma aplicação que foi desenvolvida em JADE que implementa o caso da Presa x Predador.

1. AGENTES E SISTEMAS MULTIAGENTES

1.1 AGENTES

1.1.1 Conceito

Definir agentes com um conceito único ou padrão é muito difícil, pois existem varias abordagens e ponto de vista de diferentes autores. Além disso, devido às suas mais diversas aplicações uma definição precisa torna-se cada vez mais complicada e variada.

Na definição do dicionário [LUFT 1998] um agente é uma pessoa que age por ou no lugar de outra segundo autoridade por ela outorgada – é um representante da pessoa. Seria no caso um agente humano. Estes, geralmente, realizam tarefas determinadas, são especialistas naquilo que fazem, possuem capacidades que outras pessoas não tem, tem acesso a informações relevantes para sua tarefa e as realizam a um custo bem menor do que se pessoas comuns tentasse realizar. Exemplos não faltam como agentes de seguro, agentes bancários, agentes de viagens e etc. No caso de agentes artificiais, mais precisamente agentes de software os quais são objetos de estudo, as características acima citadas é também aplicável apesar de não haver uma definição universalmente aceita. No entanto, o que se tem visto é um consenso no fato de que agentes são entidades que se adaptam a um meio, reagem a ele e provocam mudanças neste meio.

Ted Selker, pesquisador do Centro de Pesquisas da IBM em Almaden, dá uma definição interessante sobre agentes: “Um agente é um software que sabe como fazer coisas que você provavelmente faria sozinho se tivesse tempo”.

A FIPA (Foundation for Intelligent Physical Agents) [FIPA] define agente como "uma entidade que reside em um ambiente onde interpreta dados através de sensores que refletem eventos no ambiente e executam ações que produzem efeitos no ambiente. Um agente pode ser software ou hardware puro…".

Alguns pesquisadores, como Michael Wooldridge e Nick Jennings [WOOLDRIDGE 1995], adotaram duas definições gerais: noção fraca e noção forte de agentes. Na definição ou noção fraca de agentes, eles conceituam como sistemas computacionais, sendo hardware ou software, com certas propriedades tais como autonomia, habilidade social, reatividade e pró-atividade. Na noção forte de agentes, mais adotada pelos pesquisadores ligados a área de Inteligência Artificial, possui, alem das propriedades acima citadas, noções relacionadas ao comportamento humano tais como o conhecimento, a crença, a intenção e a obrigação.

Pode ser uma entidade física ou virtual, como Ferber [FERBER 1999], classifica. Física, seria alguma coisa concreta e que atue no mundo real, virtual seriam justamente entidades abstratas tais como componentes de softwares que é nosso objeto de estudo.

Algumas comparações entre objetos e agentes são inevitáveis, porém, apesar de existir semelhanças, as diferenças são muito mais evidentes. Diferenças, segundo [WEISS 1999], são o fato de agentes terem uma noção mais forte de autonomia em relação aos objetos, ser capazes de um comportamento flexível e pela característica de um sistema multiagente ser inerentemente *multi-thread*.

No geral, agentes seriam entidades de software autônomas que atuam em determinado ambientes de forma a interagir com este e com outros agentes. Além de produzir ações e percepções sem requerer intervenções humanas constantes.Numa abordagem mais aplicada a Inteligência Artificial um agente ideal teria que ser capaz de funcionar continuamente e adquirir experiências e conhecimentos acerca do ambiente que está interagindo.Ou seja, ser capaz de "aprender" e tomar decisões a partir de situações diferentes.

Pode-se dizer algumas propriedades que são essenciais para uma melhor caracterização de agente. Uma delas é a *autonomia* que o agente inteligente deve ter tomar decisões e ações importantes para a conclusão de uma tarefa ou objetivo sem a necessidade da interferência do ser humano ou qualquer outra entidade.Ou seja, ser capaz de agir independentemente com seu ambiente através de seus próprios "sensores" ou com as suas

próprias percepções com o objetivo de realizar alguma tarefa seja ela externa ou gerada por ele próprio. Operam sem a intervenção humana e tem algum tipo de controle sobre suas ações e seu estado interno.

Associado à autonomia está a *pró-atividade*, que nada mais é da capacidade que o agente deve ter de tomar iniciativas. Eles não respondem simplesmente de acordo com o meio. Têm a capacidade de exibir comportamentos baseados em objetivos.

Reatividade é a capacidade de reagir rapidamente a alterações no ambiente, ou seja, percebe o meio e responde de modo oportuno.

Um agente deve ter também *robustez* para ser capaz de tomar decisões baseando-se em informações incompletas ou escassas, lidar com erros e ter uma capacidade de adaptação ou aprendizagem através da experiência.

Para alcançar seus objetivos, agentes devem ter uma *habilidade de comunicação* que nada mais é que uma capacidade de comunicação com repositórios de informações. Seja apenas repositório de dados, seja outro agente ou seja o próprio ambiente é fundamental uma constante troca de informações.

O *raciocínio* é talvez o aspecto mais importante que distingue um agente inteligente dos outros agentes. Afirmar que um agente tem raciocínio significar dizer que ele tem a capacidade de analisar e inferir baseando-se no seu conhecimento atual e nas suas experiências. Esse raciocínio pode ser:

- Baseado em regras - onde eles usam um conjunto de condições prévias para avaliar as condições no ambiente externo.
- Baseado em conhecimento - onde eles têm à disposição grandes conjuntos de dados sobre cenários anteriores e ações resultantes, dos quais eles deduzem seus movimentos futuros.

Para Wooldridge em [WEISS 1999], uma flexibilidade de ações tais como reatividade,pró-atividade e sociabilidade, são suficientes para classificar um agente como inteligente.

Por fim, deve haver uma capacidade de cooperação: agentes inteligentes podem, e devem, trabalhar juntos para mútuo benefício na execução de uma tarefa complexa e um comportamento adaptativo, no qual possa examinar o meio externo e adaptar suas ações para aumentar a probabilidade de ser bem sucedido em suas metas.

Para garantir uma maior segurança e confiança ao usuário de que o agente vai representar fielmente seu papel com precisão é fundamental também um alto grau de confiabilidade por parte do agente. Isso é necessário para que o agente simule e ou represente da maneira mais possivelmente próxima da realidade.

Contudo, um agente não precisa ter todas essas características ao mesmo tempo. Existem agentes que possuem algumas, outros que possuem todas, o que é certo é que atualmente existe pouca concordância sobre a importância dessas propriedades e se é necessária sua obrigatoriedade para a caracterização de um agente. O consenso é que essas características tornam em muito um agente diferente de simples programas e objetos.

1.1.2 Categoria de Agentes

Para uma melhor compreensão da aplicabilidade da tecnologia de agentes, é interessante falar sobre os diversos tipos de agentes e suas mais variadas diferenças para que tenhamos uma melhor noção de utilidade no emprego de agentes. É possível fazer uma classificação de agentes de acordo com vários aspectos como quanto à mobilidade, quanto ao relacionamento inter agentes e quanto à capacidade de raciocínio.

Agentes Móveis: são agentes que tem a mobilidade como característica principal. Isto é, uma capacidade de mover-se seja por uma rede interna local (intranet) ou até mesmo pelo Web, transportando-se pelas plataformas levando dados e códigos. Seu uso tem crescido devido alguns fatos como uma heterogeneidade cada vez maior das redes e seu grande auxílio em tomadas de decisões baseadas em grandes quantidades de informação.

Agentes situados ou estacionários: são aqueles opostos aos móveis.Isto é, são fixo em um mesmo ambiente e ou plataforma. Não se movimentam em uma rede e muito menos na Web.

Agentes Competitivos: são agentes que “competem” entre si para a realização de seus objetivos ou tarefas. Ou seja, não há colaboração entre os agentes.

Agentes Coordenados ou Colaborativos: agentes com a finalidade de alcançar um objetivo maior, realizam tarefas especificas porém coordenando-as entre si de forma que suas atividades se completem.

Agentes Reativos: é um agente que reage a estímulos sem ter memória do que já foi realizado no passado e nem previsão da ação a ser tomada no futuro. Não tem representação do seu ambiente ou de outros agentes e são incapazes de prever e antecipar ações. Geralmente atuam em sociedades como uma colônia de formiga por exemplo. Baseiam-se muito também na "teoria do caos" no qual afirma que até mesmo no caos existe uma "certa organização". No caso da formiga, por exemplo, uma única dela não apresenta muita inteligência mas quando age no grupo comporta-se o todo como uma entidade com uma certa inteligência. Ou seja, a força de um agente reativo vem da capacidade de formar um grupo e construir colônias capazes de adaptar-se a um ambiente.

Agentes Cognitivos: esses, ao contrario dos agentes reativos, podem raciocinar sobre as ações tomadas no passado e planejar ações a serem tomadas no futuro. Ou seja, um agente cognitivo é capaz de "resolver" problemas por ele mesmo. Ele tem objetivos e planos explícitos os quais permitem atingir seu objetivo final. Ferber afirma que para que isso se concretize, cada agente deve ter uma base de conhecimento disponível, que compreende todo os dados e todo o *"know-how"* para realizar suas tarefas e interagir com outros agentes e com o próprio ambiente. Sua representação interna e seus mecanismos de inferência o permitem atuar independentemente dos outros agentes e lhe dão uma grande flexibilidade na forma de expressão de seu comportamento. Além disso, devido a sua capacidade de raciocínio baseado nas representações do mundo, são capazes de ao mesmo tempo memorizar situações, analisá-las e prever possíveis reações para suas ações.

1.1.3 Aplicação de Agentes Inteligentes

Segue abaixo apenas das inúmeras aplicações de agentes inteligentes:

- Agentes no processo ensino-aprendizagem

  Utilização de agentes no aprendizado e no processo de treinamento de usuários.

- Agentes na indústria

  A grande vantagem da aplicação de agentes na produção industrial é o fato de que muitos dos sistemas que atuam neste propósito são complexos e isolados. O papel dos agentes seria integrá-los e compartilhar informações entre os sistemas.

- Agentes em simulação

  Agentes podem simular situações dando um grau maior de veracidade. Tanto na área de entretenimento quanto na área de pesquisa tecnológica e militar.

- Agentes em realidade virtual

  Nessas aplicações o agente atua como um participante que auxilia e monitora certas atividades e usuários, assistindo-os ou ajudando-os quando necessário, principalmente em ambientes virtuais muito complexos.

- Agentes na prestação de serviços

  Nesse contexto os agentes irão agregar valor a certos serviços, gerenciando a informação a fim de satisfazer as necessidades dos clientes. Podemos citar por exemplo:

  - manutenção e atualização de bases de dados explorando a informação (*data mining*);
  - comércio eletrônico, onde agentes procuram preços mais convenientes ou ofertas e produtos com as especificações desejadas pelo usuário;
  - gerência de correio eletrônico e filtragens de mensagens, funcionando como um assistente pessoal;
  - gerência de produção e muitas outras aplicações ;

- Agentes em redes de computadores

  As suas funções nessas aplicações variam muito. Varia desde definição de melhores rotas, controle de manutenção de equipamentos, gerenciamento de dados até a monitoramento de atividades e gerência da própria rede.

1.2 SISTEMAS MULTIAGENTES

Dentro do contexto da Inteligência Artificial Distribuída, [TORSUN 1995] classifica-a em *Distributed Problem Solving (DPS), Multi-Agent Systems (MAS) e Parallel Artificial Inteligence (PAI).* **DPS** ou Resolução Distribuída de Problemas decompõe o

problema em módulos através de uma abordagem descendente (*top-down)* desenhado especificamente para um problema em particular, onde grande parte do raciocínio sobre a solução é inserida pelo próprio projetista. **PAI** ou Inteligência Artificial Paralela interessa-se mais por performance do que por avanços conceituais, preocupando-se principalmente em desenvolver linguagens e algoritmos de computação paralela. Por último, **MAS** ou Sistemas Multi-Agentes, caracteriza-se pela existência de agentes que interajam de forma autônoma que trabalhem juntos para resolver um determinado problema ou objetivo.

Em suma, pode-se dizer que Sistemas Multi-Agentes são sistemas constituídos de múltiplos agentes que interagem ou trabalham em conjunto de forma a realizar um determinado conjunto de tarefas ou objetivos. Esses objetivos podem ser comuns a todos os agentes ou não. Os agentes dentro de um sistema multiagente podem ser heterogêneos ou homogêneos, colaborativos ou competitivos, etc. Ou seja, a definição dos tipos de agentes depende da finalidade da aplicação que o sistema multiagente está inserido.

Os sistemas multiagentes com agentes reativos são constituídos por um grande número de agentes. Estes são bastante simples, não possuem inteligência ou representação de seu ambiente e interagem utilizando um comportamento de ação/reação. A inteligência surge conforme essas grandes trocam de interações entre os agentes e o ambiente. Ou seja, os agentes não são inteligentes individualmente, mas o comportamento global é.

Já os sistemas multiagentes constituídos por agentes cognitivos são geralmente compostos por uma quantidade bem menor de agentes se comparado aos sistemas multiagentes reativos. Estes, conforme a definição de agentes cognitivos, são inteligentes e contêm uma representação parcial de seu ambiente e dos outros agentes. Podem, portanto, comunicar-se entre si, negociar uma informação ou um serviço e planejar uma ação futura. Esse planejamento de ações é possível pois em geral os agentes cognitivos são dotados de conhecimentos, competências, intenções e crenças, o que lhes permite coordenar suas ações visando à resolução de um problema ou a execução de um objetivo.

Atualmente a pesquisa em sistemas multiagentes está interessada principalmente na coordenação das ações e comportamentos dos agentes, como eles coordenam seu conhecimento, planos, objetivos e crenças com o objetivo de tomar ações ou resolver problemas.

Segundo [JENNINGS 1998], algumas razões para o crescimento do interesse em pesquisas com sistemas multiagentes são:

- A capacidade de fornecer robustez e eficiência
- A capacidade de permitir interoperabilidade entre sistemas legados
- A capacidade de resolver problemas cujo dado, especialidade ou controle é distribuído.

Apesar de muitas vantagens, ainda segundo o mesmo artigo, sistemas multiagentes ainda possuem muitos desafios e dificuldades tanto em relação a projeto como implementação. Segue alguns abaixo:

1. Como programar, descrever, decompor e alocar problemas e sintetizar resultados com um grupo de agentes inteligentes?
2. Como permitir a comunicação e a interação entre agentes? Quais linguagens de comunicação e protocolos usar? O que e quando comunicar?
3. Como assegurar que agentes atuam coerentemente tomando decisões ou executando ações?
4. Como permitir agentes individuais atuar e raciocinar sobre ações, planos e conhecimento sobre outros agentes em ordem de coordená-los? Como raciocinar sobre o estado de seus processos coordenados?
5. Como identificar e reconciliar pontos de vistas muito diferentes e intenções conflitantes em uma coleção de agentes que tentam coordenar suas ações?
6. Como equilibrar efetivamente computação local e comunicação? Mais genericamente, como gerenciar alocação de recursos limitados?
7. Como evitar ou minimizar comportamento prejudicial do sistema, tal como um comportamento caótico?
8. Como projetar e construir sistemas multiagentes práticos? Como projetar plataformas tecnológicas e metodologias de desenvolvimento para sistemas multiagentes?

Podemos resumir em apenas três os principais desafios que um sistema multiagente possui. O primeiro diz respeito à *comunicação*. Como ela seria realizada entre os agentes e que tipo de protocolos usar? A segunda seria a *interação*: Como essa interação ocorrerá? Que linguagem os agentes devem usar para interagirem entre si e combinar seus esforços? E por último a *coordenação*: Como garantir essa coordenação entre os agentes para que haja uma coerência na solução do problema ou objetivo que estão tentando resolver?

1.3 COMUNICAÇÃO ENTRE AGENTES

A comunicação é fundamental para permitir que haja colaboração, negociação, cooperação e etc. entre entidades independentes. Em sistemas multiagentes, é necessário que a comunicação seja disciplinada para que os objetivos sejam alcançados efetiva e eficientemente, necessitando assim uma linguagem que possa ser entendida pelos outros agentes presentes no ambiente. Essa comunicação tem como principal objetivo à partilha do conhecimento com os outros agentes e a coordenação de atividades entre agentes. Ou seja, ela deve permitir que agentes troquem informações entre si e coordenem suas próprias atividades resultando sempre em um sistema coerente.

Existem diversas maneiras para agentes trocar informações uns com os outros em sistemas multiagentes.[BAKER 1997] Agentes podem trocar mensagens diretamente, chamada também por alguns autores como *comunicação direta*, podem comunicar-se através de um agente "facilitador" especial em sistema "federado" (comunicação assistida), podem também utilizar uma *comunicação por difusão de mensagens* (*broadcast)* e até utilizar o modelo de comunicação através de *blackboard* ou *quadro-negro.*

A comunicação direta, ou comunicação via troca de mensagens direta, cada agente comunica diretamente com qualquer outro agente sem qualquer intermediário. Na Figura 1.1 há um estabelecimento de uma ligação direta (ponto-a-ponto) entres os agentes através de um conjunto de protocolos que garante a chegada de mensagens com segurança. Nesse tipo de comunicação faz-se necessário que cada agente envolvido tenha conhecimento da existência dos outros agentes e da forma de como endereçar mensagens para eles. A principal vantagem deste tipo de comunicação entre agentes é o fato de não existir um agente coordenador da comunicação. Isso porque esses agentes coordenadores podem levar a um "gargalo" ou até ao bloqueio do sistema caso haja, por exemplo, um grande número de troca de mensagens. As desvantagens são o custo da comunicação que se torna grande, principalmente quando há um grande número de agentes no sistema, e a própria implementação que se torna muito complexa em comparação às outras formas de comunicação.

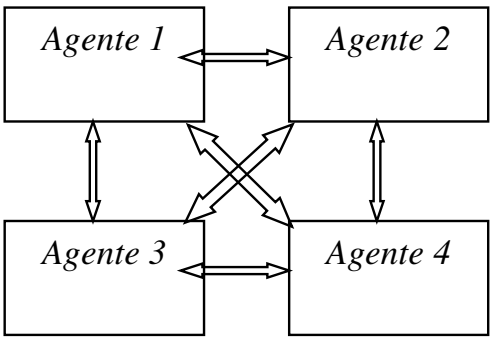

Figura 1.1 Comunicação Direta entre agentes

Na comunicação por sistemas federados [BAKER 1997] ou comunicação assistida, os agentes utilizam algum sistema ou agente especial para coordenar suas atividades. Ou seja, uma estrutura hierárquica de agentes é definida e a troca de mensagens dá-se através agentes especiais designados "facilitadores" ou mediadores (Veja a figura 1.2). Essa é uma alternativa bem popular à comunicação direta pois diminui muito o custo e a complexidade necessária aos agentes individuais na realização da comunicação. Geralmente é utilizado quando o número de agentes dentro do sistema é muito grande.

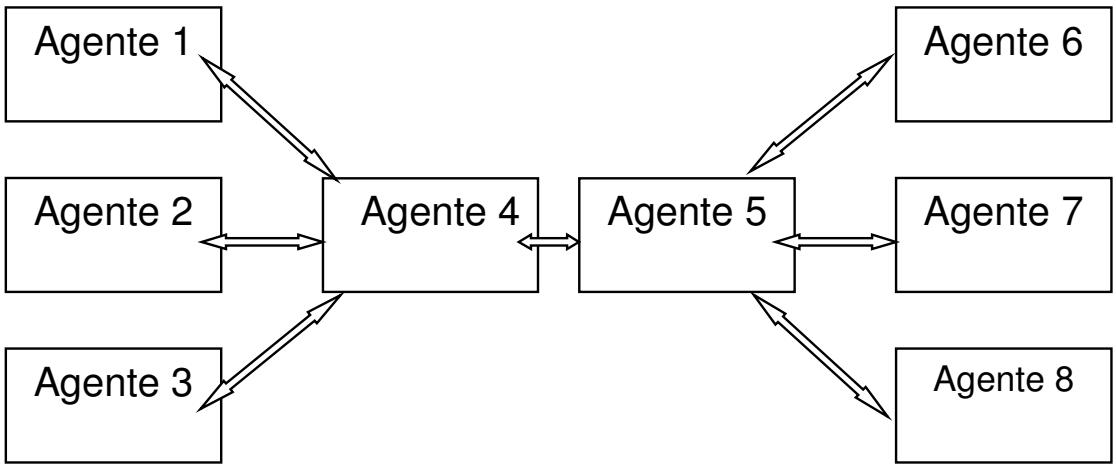

Figura 1.2. Comunicação por sistema "federado" ou comunicação assistida

A comunicação por difusão de mensagens ou broadcast geralmente é utilizada em situações onde a mensagem deve ser enviada para todos os agentes do ambiente ou quando o agente remetente não conhece o agente destinatário ou seu endereço.Em suma, todos os agentes recebem a mensagem enviada.

Comunicação por quadro-negro ou *blackboard*, segundo [BAKER 1997], é bastante usada na Inteligência Artificial como modelo de memória compartilhada.Ou seja, nada mais é que um repositório onde os agentes escrevem mensagens a outros agentes e obtêm informações sobre o ambiente.

1.4 INTERAÇÃO: Linguagens de Comunicação entre Agentes

A forma de interação que ocorre entre os agentes é um fator muito importante na integração destes. As linguagens de comunicação e sua expressividade definem a capacidade de comunicação de cada agente. Ela deve ser universal e partilhada por todos os agentes, ser concisa e ter um número limitado de primitivas de comunicação. Como modelo de comunicação de agentes são usadas a comunicação humana e a teoria dos atos comunicativos (*Speech Act Theory)* [MENESES 2001]. Esta teoria usa o conceito de performativas para permitir conduzir suas ações. Ricardo Gudwin [GUDWIN 2003] define certas características desta teoria:

- Derivada da análise lingüística da comunicação humana
- Com uma linguagem, um falante de uma língua não somente efetua uma declaração, mas realiza uma ação
- Mensagens são ações ou atos comunicativos

Gudwin afirma que "os atos comunicativos são interpretados a partir da mensagem do contexto..." e "...nem sempre essa interpretação é obvia...". Em outras palavras, significa que esses atos comunicativos são sujeitos a interpretações dúbias que podem ter significados diferentes de acordo com o ponto de vista.

Dificuldades [GUDWIN 2003]:

- "Saia da minha frente!" (Comando)
- "Por favor, saia da minha frente" (Pedido)

- "Você poderia sair da minha frente?" (Pergunta)
- "Eu gostaria que você saísse da minha frente" (Informação)

Por isso se faz necessário sempre deixar explicito o ato comunicativo relacionado à mensagem na comunicação entre agentes.

1.4.1 KQML

KQML ou *Knowledge Query and Manipulation Language* é uma linguagem e um protocolo de comunicação de alto nível para troca de mensagens independente de conteúdo e da ontologia aplicável. Ela foi desenvolvida pelo KSE – "*Knowledge Sharing Effort*" para servir ao mesmo tempo como um formato de mensagem e um protocolo de gerenciamento de mensagens. Ela não se preocupa muito com o conteúdo da mensagem mas sim com a especificação da informação necessária à compreensão do conteúdo.

Segundo [GUDWIN 2003], o significado de performativas reservadas e padrões no KQML é pouco claro: "...normalmente estão associadas à intuição...". Outros problemas e dificuldades sobre KQML são:

- Ambigüidade e termos vagos
- performativas com nomes inadequados – *"...algumas performativas têm nomes que não correspondem diretamente ao ato comunicativo a ela associado.*" [GUDWIN 2003]
- Falta de performativas – *"Alguns atos comunicativos não estão representados entre as performativas disponíveis."* [GUDWIN 2003]

Devido aos problemas acima citados alguns autores afirmam que KQML provavelmente será substituído por FIPA-ACL. Na figura 1.3, temos uma estrutura básica de uma mensagem no formato KQML.

```
(KQML-performative
        :sender <word>
        :receiver <word>
        :reply-with <word>
        :language <word>
        :ontology <word>
          :content <expression>
        …… )
```

Figura 1.3. Formato básico de uma mensagem KQML [WEISS 1999]

1.4.2 FIPA-ACL

A FIPA-ACL é uma linguagem, como o KQML, baseada em ações de fala. A sua sintaxe é bastante semelhante ao KQML, porém o conjunto de performativas (atos comunicativos) é diferente. Sua especificação consiste de um conjunto de tipos de mensagens e descrições dos efeitos da mensagem sobre os agentes que a enviam e sobre o que a recebem. Possui uma semântica definida precisamente com uma linguagem de descrição de semântica.

De acordo com [MENESES 2001], KQML tem sido muito criticada por usar o termo performativo para se referir às primitivas de comunicação. Em FIPA-ACL, essas primitivas são chamadas de ações ou atos comunicativos (*communicative acts*).

Ricardo Gudwin [GUDWIN 2003] prevê o futuro da FIPA-ACL: "deve vir a substituir o KQML, pois resolve a maioria dos problemas criticados por diferentes autores na concepção do KQML.".

Na figura 1.4 vemos uma estrutura de uma mensagem em FIPA – ACL, com as partes que compõem uma mensagem básica.

```
(communicative act
        :sender <valor>
        :receiver <valor>
        :content <valor>
        :language <valor>
        :ontology <valor>
        :conversation-id<valor>
        …)
```

Figura 1.4. Formato de uma mensagem FIPA-ACL (Semelhante ao KQML)

1.5 COORDENAÇÃO ENTRE AGENTES

Gerhard Weiss [WEISS 1999] afirma que “coordenação é uma característica fundamental para um sistema de agentes que executam alguma atividade em um ambiente compartilhado”.

A coordenação está muito relacionada com o compartilhamento de conhecimento entre os agentes, sendo, seu principal objetivo, tornar as ações individuais de cada agente coordenadas para se atingir o objetivo final do sistema multiagente. Além disso, há uma preocupação com a coerência de modo a se discutir como o sistema multiagente por completo se comporta enquanto está resolvendo o problema. O principal motivo para uma preocupação maior com a coordenação entre agentes é o fato de que um só agente, dentro de um sistema multiagente, não terá informação ou capacidade suficiente para resolver muitos dos problemas - muitos dos objetivos não podem ser atingidos por agentes agindo isoladamente.

Assim, coordenação seria a capacidade de esses agentes trabalharem em conjunto e combinar seus objetivos de forma a concluírem o objetivo final do sistema. Geralmente para uma cooperação ser bem sucedida, cada agente deve manter um “modelo” dos outros agentes e também desenvolver um modelo de interações futuras ou possíveis.Ela pode ser dividida em cooperação e negociação.

Segundo [WEISS 1999]: “Negociação é a coordenação entre agentes antagônicos ou simplesmente egoístas (*self-interested*).” Ou seja, a negociação está relacionada à coordenação de agentes competitivos. Geralmente são usados protocolos de negociação para determinar as regras de negociação e são definidos os conjuntos de atributos sobre os quais se pretende chegar a um acordo.

Cooperação, como afirma Gerhard Weiss em [WEISS 1999], é a “...coordenação entre agentes não antagônicos...”. Ou seja, uma coordenação com agentes que não possuem objetivos conflitantes geralmente é chamada de cooperação. Neste caso, agentes cooperativos auxiliam-se mutuamente nem que para isso provoquem custos individuais. O papel da coordenação é agir de forma que conjunto de agentes realize suas tarefas como um “trabalho de equipe” visando sempre o objetivo final do sistema.

1.6 AMBIENTES MULTIAGENTES

Ambientes de sistemas multiagentes [WEISS 1999]:

- Provêm uma infra-estrutura especificando protocolos de comunicação e interação.
- São geralmente abertos e não possuem apenas um projetista – não são centralizados
- Contêm agentes que são autônomos e distribuídos, e podem ser competitivos ou cooperativos.

Propriedade	**Valores**
Autonomia de Projeto	*Plataforma/Protocolo de Interação/Linguagem/Arquitetura Interna*
Infra-estrutura de Comunicação	*Memória Compartilhada (blackboard) ou Baseada em Mensagens, Orientada à Conexão ou Não Orientada à Conexão(e-mail), Ponto-a-Ponto,Multicast ou Broadcast, Síncrono ou Assíncrono*
Diretório de Serviço (Directory Service)	*White pages, Yellow pages*
Protocolo de Mensagens	*KQML, FIPA-ACL, HTTP OU HTML,OLE , CORBA*
Serviço de Mediação	*Baseado em Ontologia ou Transações*
Serviço de Segurança	*Timestamps/Autenticação*
Operações de Suporte	*Armazenamento/Redundância/Restauração/ Accounting*

Tabela 1.1 Características de um Ambiente Multiagente [WEISS 1999]

Por uma questão didática, pode-se dividir os ambientes de desenvolvimento de sistemas multiagentes em plataformas de agentes móveis e plataformas de agentes não móveis.

As plataformas para agentes móveis, como o próprio nome já diz, fornecem todo um *framework e* ferramentas que ajudam na criação de agentes móveis juntamente com uma estrutura que facilita a sua implementação. Já as plataformas para agentes não móveis são exatamente as que fornecem ferramentas que ajudam na criação de agentes inteligentes. Muitas delas são específicas, isto é, os agentes criados por elas são específicos a uma determinada área. Isso não significa dizer que uma plataforma classificada como plataforma para agentes não-móveis não possa criar agentes móveis. Pelo contrário, muitos dos agentes criados por esses ambientes possuem até uma certa mobilidade só que muito mais simples e com funcionalidades mais primitivas do que os agentes criados por plataformas especialmente dedicadas para agentes móveis.

Segue abaixo alguns exemplos de plataformas para agentes não móveis:

⇒ **ZEUS** [BIGUS 2001][ZEUS 2003]

- Desenvolvido pela British Telcom
- Framework para o desenvolvimento de agentes colaborativos ou cooperativos
- Baseado em JAVA
- Possui três componentes principais: *The Agent Component Library/The Agent Building Tools / The Visualisation Tools*
- Usa KQML
- Site Oficial http://more.btexact.com/projects/agents/zeus/index.htm

⇒ **FIPA – OS** [BIGUS 2001]

- Implementação *open source*
- Duas versões Standart FIPA e MicroFIPA – OS (especifico para dispositivos portáteis)
- Baseado em JAVA
- Site Oficial http://fipa-os.sourceforge.net

⇒ **MICROSOFT AGENT** [MSAGENT 2003]

- Plataforma para desenvolvimento de agentes de interface que auxiliam o usuário na realização de suas tarefas
- Uso de personagens como assistentes interativos
- Requer Microsoft Windows, Internet Explorer e ActiveX, sendo portanto, dependente de plataforma Microsoft
- Site Oficial Programável em linguagens Microsoft (Visual C++, Visual Basic, etc)
- Suporte a .NET
- Site oficial http://www.microsoft.com/msagent/

1.7 PADRÃO FIPA

Foundation for Intelligent Physical[1] *Agents* ou simplesmente FIPA, é uma fundação sem fins lucrativos direcionada à produção de padrões para a interoperabilidade de agentes heterogêneos e interativos e sistemas baseados em agentes. Ela foi fundada em 1996 em Genebra. A sua missão básica é facilitar a interligação de agentes e sistemas multiagentes entre múltiplos fornecedores de ambientes. A missão oficial é: "A promoção de tecnologia e especificações de interoperabilidade que facilitem a comunicação entre sistemas de agentes inteligentes no contexto comercial e industrial moderno." [FIPA 2003]. Em suma, interoperabilidade entre agentes autônomos.

A FIPA é composta por companhias e organizações membro que se encontram em reuniões para discutir e produzir padrões para a tecnologia de agentes. Essas companhias membro da FIPA organizam-se em:

- Comitês técnicos *(Technical Comittees* - TC) – Responsáveis pelo trabalho técnico, criando ou alterando as especificações existentes. Trabalham na arquitetura (agente – descrição, localização, permissões e obrigações), na conformidade (gerando perfis conformes às especificações da FIPA), nas

[1] A organização tem a palavra *physical* por motivos históricos. Antes a FIPA pretendia padronizar agentes robóticos também.

ontologias (desenvolvendo e adaptando ontologias existentes para serem usadas por agentes FIPA) e na semântica (desenvolvendo um *framework* de semânticas para conversações, interações e comportamentos sociais).

- Grupos de trabalho (*Working Groups* - WG) – Cria documentos informativos ou propostas de alterações às especificações existentes para os TCs apropriados. Também foram criados para executar testes de interoperabilidade, "testes de campo" e aplicações dando um "feedback" para as especificações.
- Comissão de diretores (*Board of Directors* - BD) – Responsáveis pela gestão da FIPA, definindo políticas e procedimentos que a FIPA adota para os ambientes de desenvolvimentos de agentes.
- Comissão de arquitetura (*Architecture Board* -AB) - Responsáveis por assegurar a consistência e precisão do trabalho técnico a ser desenvolvido. Aprova e supervisiona os planos de trabalho dos TCs e dos WGs.

Periodicamente, oficialmente três vezes por ano, são realizadas reuniões para permitir que haja uma maior colaboração por parte de pesquisadores e desenvolvedores na discussão e produção das especificações da FIPA. Isso permite também que o BD e AB da FIPA se dirijam aos membros e que estes possam escolher e agrupar-se nos TCs e WGs que lhes interesse.

Segundo dados encontrados no site oficial [FIPA 2003], antes da FIPA havia cerca de 60 sistemas proprietários de agentes em competição, sendo que a maioria destes eram sistemas "fechados" e incompatíveis. Fatos tais que atrasaram o desenvolvimento da tecnologia de agentes. Logo, fica claro que a necessidade de padronização torna-se indispensável.

Assim, a FIPA se apóia basicamente em duas concepções principais. A primeira é que o tempo para alcançar um consenso e para completar a padronização não deve ser longo para que não impeça o progresso nessa área de pesquisa. A segunda é que apenas os comportamentos externos dos componentes é que podem ser especificados, deixando detalhes de implementação e arquiteturas internas para os desenvolvedores de agentes.

Algumas das linguagens de desenvolvimento de agentes que estão relacionadas com a FIPA são: **Agent Development Kit[2], FIPA-OS, Lightweight Extensible Agent Plataform[3] (LEAP), ZEUS e o próprio JADE.**

Figura 1.5 Alguns membros industriais da FIPA

[2] Mais informações sobre o *Agent Development Kit* podem ser encontradas no site http://edocs.bea.com/manager/mgr20/pguide/overview.htm

[3] Site Oficial do Projeto LEAP – http://leap.crm-paris.com

2. JADE

Antes de falarmos em JADE propriamente dito, daremos uma breve introdução sobre a linguagem Java pelo fato de JADE ter sido totalmente desenvolvido e baseado nessa linguagem. Java possui diversos mecanismos nativos à linguagem que simplificam a programação de sistemas multiagentes, como por exemplo, concorrência e a comunicação entre objetos distribuídos.

2.1 JAVA

A linguagem Java foi criada pela Sun Microsystems em 1995 para desenvolvimento de programas em ambientes heterogêneos ligados em rede. Seu o objetivo inicial era ser utilizada em sistemas isolados com quantidade mínima de memória (dispositivos eletrônicos para o consumidor final). Com a explosão da internet e da *World Wide Web*, a linguagem Java gerou um interesse maior por parte da comunidade pois se percebeu um grande potencial da linguagem na criação de páginas *web* com conteúdo dinâmico [DEITEL 2000]. Atualmente, Java é utilizada em diversas áreas: desde o desenvolvimento de aplicativos corporativos de larga escala, como aplicativos para dispositivos portáteis chegando até a programação de agentes que é nosso foco principal.

Java é composto basicamente por três elementos: uma linguagem orientada a objetos, um conjunto de bibliotecas e uma máquina virtual no qual os programas são interpretados.

O ambiente de execução foi construído com base em uma máquina virtual. O código fonte Java é compilado para um código intermediário em bytes (*bytecodes)* e em

seguida pode ser interpretado em qualquer implementação da máquina virtual Java (*Java Virtual Machine – JVM).* Desta forma, depois de compilados para *bytecodes*, os programas Java podem ser executados sem alterações em qualquer plataforma para a qual existir um ambiente de execução Java. Além disso, muitos mecanismos foram integrados em Java tais como, *threads,* comunicação entre objetos e tratamento de eventos. Essas características não são conceitos novos, mas em conjunto fornecem um grande poder à linguagem principalmente no desenvolvimento de sistemas multiagentes.

Java é caracterizado por ser:

- **Simples -** Os desenvolvedores basearam-se na linguagem de programação C++, mas removeram muitas das características de orientação a objeto que são raramente usadas, tornando Java muito mais simples.
- **Orientado a Objeto –** Java suporta a escrita de um software utilizando o paradigma orientado a objeto. Programação orientada a objetos é baseada na modelagem do mundo real em termos de componentes de software denominados objetos. Um objeto consiste de dados e operações, estas chamadas de métodos que podem ser realizadas sobre os dados que os encapsulam e os protegem sendo o único meio de mudar o estado do dado. Outro aspecto da orientação a objeto é a herança. Objetos podem utilizar características de outros objetos sem ter que reproduzir a funcionalidade daquele objeto (reusabilidade de código).
- **Distribuído –** Java é especificamente projetado para trabalhar dentro um ambiente de redes, possuindo uma grande biblioteca de classes para comunicação entre sistemas heterogêneos e distribuídos.
- **Interpretado -** Quando um compilador Java traduz um arquivo fonte de classe em Java para *bytecodes*, estes *bytecodes* podem rodar em qualquer máquina que possua uma máquina virtual Java. Isto permite que o código em Java possa ser escrito independente da plataforma, e elimina a necessidade de compilar e rodar no cliente uma vez os *bytecodes* não são compilados e sim interpretados.
- **Robusto -** Robustez significa que Java impõe restrições para evitar erros do desenvolvedor. Estas restrições incluem aquelas com relação aos tipos de dados e ao uso de ponteiros e acesso direto à memória.
- **Alta Performance -** Devido aos *bytecodes* de Java serem interpretados, a performance às vezes não é tão rápida quanto se fosse uma compilação direta.

A compilação em Java inclui uma opção para traduzir os *bytecodes* para código nativo de determinada plataforma. Isto pode dar a mesma eficiência que uma compilação tradicional. De acordo com teste da Sun Microsystems, a performance deste *bytecode* para tradução em código de máquina é comparável a compilação direta de programas em C ou C++.

- **Multithreaded -** Java é uma linguagem que pode ser utilizada para criar aplicações nas quais múltiplas linhas de execução podem ocorrer ao mesmo tempo, baseando-se em um sistema de rotinas que permite para múltiplos *threads*.

Em relação ao desenvolvimento de sistemas multiagentes, [BIGUS 2001] relaciona algumas características importantes de Java:

Autonomia – Para que um programa seja autônomo, ele deve ser um processo ou uma *thread* separada. Aplicações Java são processos separados e como tais podem executar por um longo período de tempo. Além disso, é possível também implementar um agente como uma *thread* separada. A comunicação pode ser feita através de *sockets* ou invocação remota de métodos (*Remote Method Invocation* - RMI). Outro problema relacionado à autonomia é como um agente sabe que alguma coisa mudou. A forma mais natural é através notificação por eventos. Java disponibiliza um sistema de tratamento de eventos, utilizado nos sistemas de janelas AWT (*Abstract Windowing Toolkit).*

Inteligência - A inteligência em agentes abrange tanto código lógico procedural ou orientado a objetos quanto capacidades sofisticadas de raciocínio e aprendizagem. Embora Lisp e Prolog sejam linguagens consagradas para este tipo de aplicação, vários trabalhos vêm sendo feitos em C e C++, e, portanto, podem ser implementados também em Java. Representações de conhecimento padrões em Inteligência Artificial como *frames*, redes semânticas e regras *if-then* podem ser facilmente e naturalmente implementadas usando Java.

Mobilidade – A mobilidade é facilitada em Java pela portabilidade do *bytecode* e dos arquivos JAR[4] (várias classes agrupadas em um único arquivo para facilitar a distribuição dos programas). Em Java é possível enviar os códigos pela rede e executar em outra máquina, como por exemplos *applets*. Um dos requisitos para programas com mobilidade é a habilidade de salvar o estado do processo em execução, despachá-lo e então restaurar o processo onde

[4] Arquivos JAR (Java Archive) são arquivos baseados no formato ZIP que são usados para agregar muitos arquivos em um só. Eles são fundamentais no desenvolvimento de aplicações de médio e grande porte, agrupando em um único arquivo todos os arquivos de classes de um projeto. Mais informações em: http://java.sun.com/products/jdk/1.2/docs/guide/jar/jarGuide.html

quer que ele tenha sido entregue, mesmo que agora esteja executando em um sistema diferente. Uma vez que a JVM provê um ambiente de computação padrão para processos executando Java, irá prover também uma máquina virtual homogênea que permite que agentes Java se movam entre sistemas de hardware heterogêneos.

Java possui bibliotecas pré-definidas que estão prontas para ser usadas, seja por instanciação direta seja por herança. Uma biblioteca é um conjunto de arquivos ".class" (*bytecodes)*. Cada biblioteca corresponde a um pacote (*package)* que serve para agrupar classes relacionadas.

Por atuar em sistemas distribuídos e conseqüentemente possíveis máquinas diferentes, é necessário uso de tecnologias para que a interoperabilidade entre ambientes heterogêneos seja possível: *Remote Method Invocation (RMI)* é uma das disponíveis que Java utiliza. Invocação remota de métodos (RMI) é um conjunto de classes e interface em Java que encapsulam vários mecanismos de troca de dados, a fim de simplificar a execução de chamadas de métodos remotamente localizados em espaços de endereçamento diferentes, potencialmente em alguma outra JVM. Um objeto RMI é um objeto remoto em Java cujos métodos podem ser chamados de outra JVM, normalmente usando a rede.

2.2 O QUE É JADE

Jade[5] (***J**ava **A**gent **DE**velopment framework)* é um ambiente para desenvolvimento de aplicações baseada em agentes conforme as especificações da FIPA[6] (*Foundation for Intelligent Physical Agents*) para interoperabilidade entre sistemas multiagentes totalmente implementado em Java. Foi desenvolvido e suportado pelo CSELT da Universidade de Parma na Itália. É *open source* (LGPL[7]) Segundo [JADE 2003], o principal objetivo do Jade é simplificar e facilitar o desenvolvimento de sistemas multiagentes garantindo um padrão de interoperabilidade entre sistemas multiagentes através de um abrangente conjunto de agentes de serviços de sistema, os quais tanto facilitam como possibilitam a comunicação entre agentes, de acordo com as especificações da FIPA: serviço

[5] *Jade* é uma marca registrada do TILAB (http://www.telecomitalialab.com) anteriormente conhecido com CSELT. Foi desenvolvido pelo TILAB juntamente com o AOT (http://aot.ce.unipr.it) com a permissão do TILAB.
[6] Veja em http://www.fipa.org
[7] LGPL ou Lesser General Public License. Mais informações podem ser encontradas em http://www.opensource.org/licenses/lgpl-license.php

de nomes (*naming service*) e páginas amarelas (*yellow-page service)*, transporte de mensagens, serviços de codificação e decodificação de mensagens e uma biblioteca de protocolos de interação (padrão FIPA) pronta para ser usada. Toda sua comunicação entre agentes é feita via troca de mensagens. Além disso, lida com todos os aspectos que não fazem parte do agente em si e que são independentes das aplicações tais como transporte de mensagens, codificação e interpretação de mensagens e ciclo de vida dos agentes. Ele pode ser considerado como um "*middle-ware*" de agentes que implementa um *framework* de desenvolvimento e uma plataforma de agentes. Em outras palavras, uma plataforma de agentes em complacência com a FIPA e um pacote, leia-se bibliotecas, para desenvolvimento de agentes em *Java.*

De acordo com [BELLIFEMINE 2003], *Jade* foi escrito em *Java* devido a características particulares da linguagem particularmente pela programação orientada a objeto[8] em ambientes distribuídos heterogêneos. Foram desenvolvidos tanto pacotes Java com funcionalidades prontas pra uso quanto interfaces abstratas para se adaptar de acordo com a funcionalidade da aplicação de agentes.

2.3 CARACTERÍSTICAS DE JADE

Seguem abaixo algumas características que Jade oferece para a programação de sistemas multiagentes:

- Plataforma distribuída de agentes - JADE pode ser dividida em vários "*hosts*" ou máquinas (desde que eles possam ser conectados via RMI). Apenas uma aplicação Java e uma *Java Virtual Machine* é executada em cada *host*. Os agentes são implementados como *threads* Java e inseridos dentro de repositórios de agentes chamados de containeres (*Agent Containers)* que provêm todo o suporte para a execução do agente.
- Gui ou *Graphical User Interface* – Interface visual que gerencia vários agentes e containeres de agentes inclusive remotamente.

[8] Mais informações sobre Java e Programação Orientação a Objeto pode ser encontrada no livro Core Java 2 – Fundamentos – Volume 1 Cay S. Horstmann & Gary Cornell

- Ferramentas de *Debugging* - ferramentas que ajudam o desenvolvimento e depuração de aplicações multiagentes baseados em JADE.
- Suporte a execução de múltiplas, paralelas e concorrentes atividades de agentes - através dos modelos de comportamentos (*Behaviours*).
- Ambiente de agentes complacente a FIPA – No qual incluem o sistema gerenciador de agentes (AMS - *Agent Management System*), o diretório facilitador (DF - *Directory Facilitator*) e o canal de comunicação dês agentes (ACC - *Agent Communication Channel*). Todos esses três componentes são automaticamente carregados quando o ambiente é iniciado.
- Transporte de mensagens – Transporte de mensagens no formato FIPA-ACL dentro da mesma plataforma de agentes.
- Biblioteca de protocolos FIPA - Para interação entre agentes JADE dispõe uma biblioteca de protocolos prontos para ser usados.
- Automação de registros - Registro e cancelamento automático de agentes com o AMS fazendo com que o desenvolvedor se abstraia disso.
- Serviços de nomes (Naming Service) em conformidade aos padrões FIPA: na inicialização dos agentes, estes obtêm seus GUID (Globally Unique Identifier) da plataforma que são identificadores únicos em todo o ambiente.
- Integração - Mecanismo que permite aplicações externas carregarem agentes autônomos JADE.

Além das características acima citadas que por si só já facilitam muito o desenvolvimento de sistemas multiagentes, Jade possui também algumas ferramentas muito úteis que simplificam a administração da plataforma de agentes e o desenvolvimento de aplicações. Essas ferramentas serão abordadas com mais detalhes no capítulo 3 e podem ser encontradas no pacote *jade.tools*.

2.4 JADE E FIPA

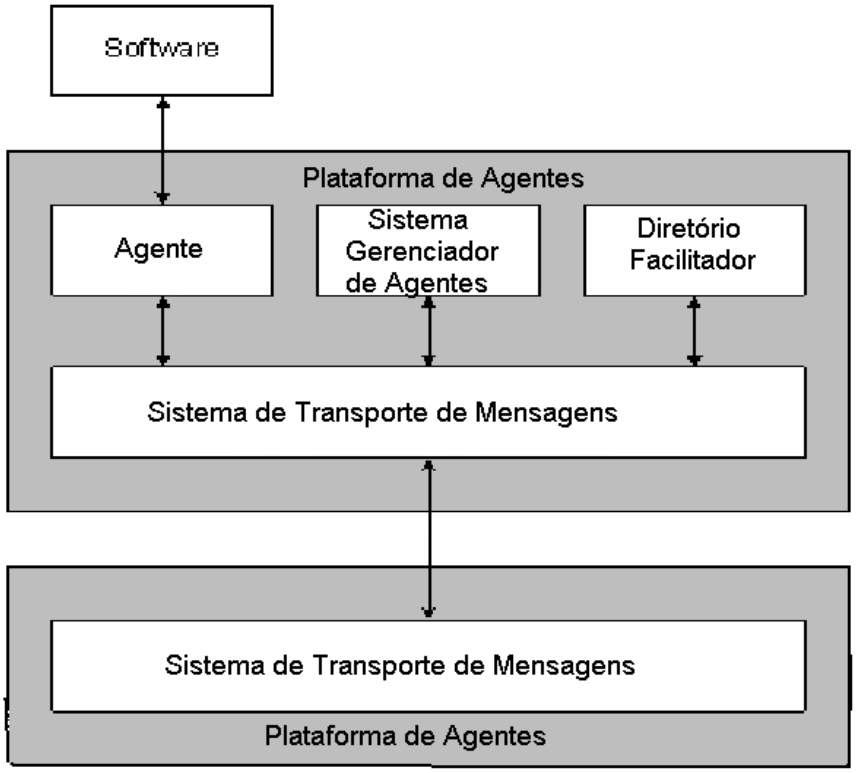

Figura 2.1. Modelo padrão de plataforma de agentes definido pela FIPA

O modelo de plataforma padrão especificado pela FIPA, que pode ser visto na Figura 2.1, é composto pelos seguintes estruturas definidas abaixo:

O agente (*Agent),* parte superior esquerda da Figura 2.1, é o agente propriamente dito cujas tarefas serão definidas de acordo com o objetivo da aplicação. Encontra-se dentro da plataforma de agentes (*Agent Platform)* e realiza toda sua comunicação com agentes através de troca de mensagens e relacionando-se com aplicação externa (software).

O sistema gerenciador de agentes ou *Agent Management System (AMS)*, parte superior central da Figura 2.1, é o agente que supervisiona o acesso e o uso da plataforma de agentes. Apenas um AMS irá existir em uma plataforma. Ele provê guia de endereços (*white-pages)* e controle de ciclo-de-vida, mantendo um diretório de identificadores de agentes (*Agent Identifier* - AID[9]) e estados de agentes. Ele é o responsável pela autenticação de

[9] AID ou *Agent Identifier* é uma classe de JADE que atribui identificadores únicos aos agentes. Será discutida no tópico 2.10 Particularidades

agentes e pelo controle de registro. Cada agente tem que se registrar no AMS para obter um AID válido.

O diretório facilitador (*Directory Facilitator - DF),* localizado na parte superior direita da Figura 2.1, é o agente que provê o serviço de páginas amarelas (*yellow-pages)* dentro da plataforma.

Na parte inferior da Figura 2.1 temos o sistema de transporte de mensagens ou *Message Transport System*, também conhecido como canal de comunicação dos agentes (*Agent Communication Channel – ACC).* Ele é o agente responsável por prover toda a comunicação entre agentes dentro e fora da plataforma. Todos os agentes, inclusive o AMS e o DF, utilizam esse canal para a comunicação.

JADE cumpre totalmente com essa arquitetura especificada. Sendo que, no carregamento da plataforma JADE, o AMS e o DF são criados e o ACC é configurado para permitir a comunicação através de mensagens.

Logo, em relação a FIPA, JADE abstrai ao programador muito das especificações dela como:

- Não há a necessidade de implementar a plataforma de agentes: o sistema gerenciador de agentes **(AMS) ,** o diretório facilitador **(DF)** e canal de comunicação dos agentes (**ACC)** são carregados na inicialização do ambiente.
- Não há a necessidade de implementar um gerenciamento de agentes: um agente é registrado na plataforma no seu próprio construtor, recebendo nome e endereço, sem falar na classe *Agent* que oferece acessos simplificados a serviços no DF.
- Não há necessidade de implementar transporte de mensagens e *parsing (“analisar gramatical” das mensagens)*: isto é automaticamente feito pelo ambiente na troca de mensagens.
- Protocolos de interação só podem ser herdados por meio de métodos específicos

2.5 ARQUITETURA INTERNA DO JADE

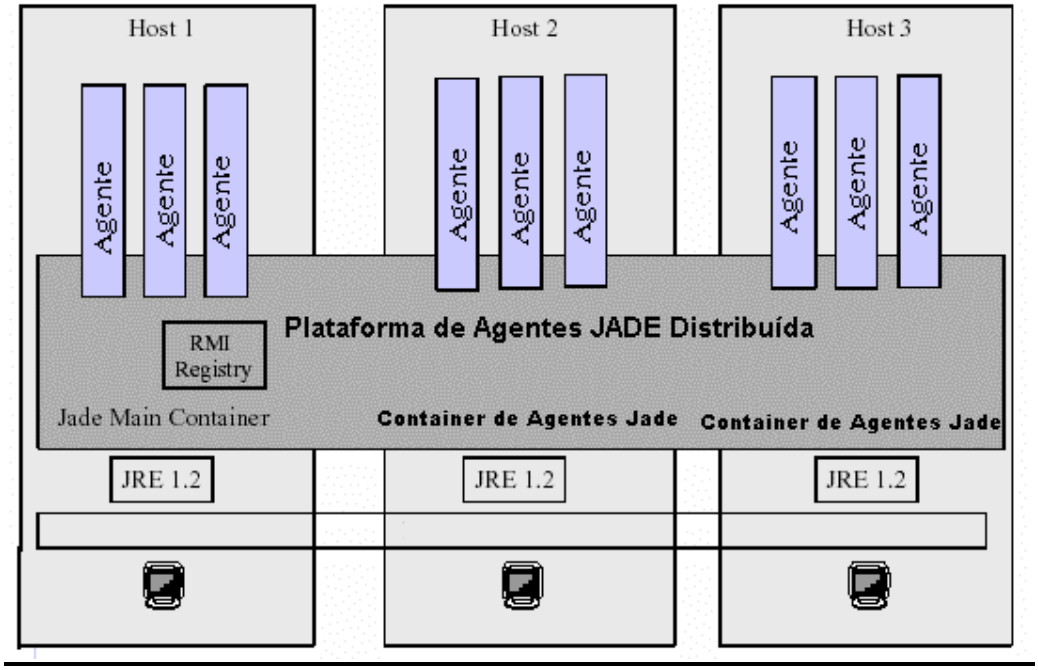

Figura 2.2 Plataforma de Agentes JADE distribuída em vários containeres.

A arquitetura da plataforma JADE é baseada na coexistência de várias máquinas virtual Java (*Java Virtual Machine* - JVM) podendo ser distribuída por diversas máquinas independente de sistema operacional que cada uma utiliza. Na Figura 2.2 temos uma visão distribuída da plataforma de agentes JADE dividida em 3 *hosts.* Em cada *host* temos uma JVM (Na Figura 2.2, JRE 1.2 significa *Java Run-time Enviroment,* versão 1.2*)* para enfatizar o conceito de independência de plataforma. Em cada JVM temos basicamente um container de agentes que fornece um ambiente completo para execução destes agentes, além de permitir que vários agentes (cada agente é representado na Figura 2.2 pelo quadro colorido Agente*)* possam rodar concorrentemente no mesmo processador (*host).* Ou seja, durante a execução deve existir uma JVM por processador, sendo possível vários agentes por JVM. A comunicação entre JVMs é feita através de invocação remota de métodos (*Remote Method Invocation ou RMI)* de Java.

O *main-container,* localizado no *host 1* da Figura 2.2, é o container onde se encontra o AMS, o DF e registro RMI (*Remote Method Invocation Registry).* Esse registro RMI nada mais é que um servidor de nomes que Java usa para registrar e recuperar referências a objetos através do nome. Ou seja, é o meio que JADE usa em Java para manter as referências aos outros containeres de agentes que se conectam a plataforma.

Os outros containeres de agentes conectam ao *main-container* fazendo com que o desenvolvedor fique abstraído da separação física dos *hosts*, caso exista, ou das diferenças de plataformas dos quais cada *host* possam ter. Essa abstração é ilustrada na Figura 2.2 na parte central na qual a plataforma JADE integra todos os três *hosts* atuando como um elo de ligação e provendo um completo ambiente de execução para qualquer conjunto de agentes JADE.

2.6 O AGENTE EM JADE

FIPA nada especifica sobre as estruturas internas dos agentes, fato que foi uma escolha explícita em conforme com a opinião de que a interoperabilidade pode ser garantida apenas especificando os comportamentos externos dos agentes (Ex: ACL, protocolos, linguagens de conteúdo e etc.) [FIPA 2003]. Para o Jade, um agente é autônomo e independente processo que tem uma identidade e requer comunicação com outras agentes, seja ela por colaboração ou por competição, para executar totalmente seus objetivos [JADE 2003].

Em outras palavras, pode-se concluir que Jade é absolutamente neutro no que diz respeito à definição de um agente. Ou seja, ele não limita ou especifica que tipo de agente pode ser construído pela plataforma.

Em termos mais técnicos um agente em Jade é funciona como uma *"thread*" que emprega múltiplas tarefas ou comportamentos e conversações simultâneas. Esse agente Jade é implementado como uma classe Java chamada *Agent* que está dentro do pacote *jade.core*. Essa classe *Agent* atua como uma super classe para a criação de agentes de software definidos por usuários. Ela provê métodos para executar tarefas básicas de agentes, tais como:

- Passagens de mensagens usando objetos *ACLMessage (*seja direta ou *multicast*);

- Suporte completo ao ciclo de vida dos agentes, incluindo iniciar ou carregar, suspender e "matar" (*killing)* um agente;
- Escalonamento e execução de múltiplas atividades concorrentes;
- Interação simplificada com sistemas de agentes FIPA para a automação de tarefas comuns de agentes (registro no DF , etc).

JADE também provê suporte ao desenvolvimento de agentes móveis. Devido a sua própria arquitetura, os agentes podem migrar e clonar-se entre os containeres (será discutida na seção 2.9 Interoperabilidade). Além disso, disponibiliza uma biblioteca (*jade.domain.mobility)* que contém a definição de ontologias para a mobilidade em JADE, vocabulário com uma lista de símbolos usados e todas as classes Java que implementam essas ontologias.

Do ponto de vista do programador, um agente JADE é simplesmente uma instância da classe *Agent*, no qual os programadores ou desenvolvedores deverão escrever seus próprios agentes como subclasses de *Agent*, adicionando comportamentos específicos de acordo com a necessidade e objetivo da aplicação, através de um conjunto básico de métodos, e utilizando as capacidades herdadas que a classe *Agent* dispõe tais como mecanismos básicos de interação com a plataforma de agentes (registro, configuração, gerenciamento remoto, etc).

Na Figura 2.3 temos uma descrição de uma arquitetura interna de um agente genérico em JADE. Na parte superior temos os comportamentos ativos do agente que representariam as ações/intenções que cada agente tem. O modelo computacional de um agente em JADE é multitarefa, onde tarefas (ou comportamentos) são executadas concorrentemente. Cada funcionalidade ou serviço provido por um agente deve ser implementado como um ou mais comportamentos (Mais detalhes sobre Comportamentos/*behaviours* na seção 2.7) . Note que esses comportamentos podem ser vários uma vez JADE permite uma variedade de comportamentos em um mesmo agente.

Na parte inferior esquerda da Figura 2.3, temos uma fila privativa de mensagens ACL. Todo agente JADE tem essa fila onde decide quando ler as mensagens recebidas e quais mensagens ler.

Ao centro, temos o escalonador de comportamentos e o gerenciador do ciclo de vida. O primeiro é responsável por escalonar a ordem de execução dos comportamentos, se por exemplo seguirão alguma ordem de precedência ou não. O gerenciador de ciclo de vida é

o controlador do estado atual do agente. Como uma característica importante do modelo de agentes JADE é autonomia., cada agente possui a autonomia implementada pela possibilidade de controlar completamente sua *thread* de execução. O gerenciador de ciclo de vida é meio que os agentes utilizam para determinar seu estado atual (ativo, suspenso, etc). No lado direito da figura, temos os recursos de agentes dependentes da aplicação. Nesse local serão armazenadas as crenças e capacidades que o agente adquiriu na execução da aplicação.

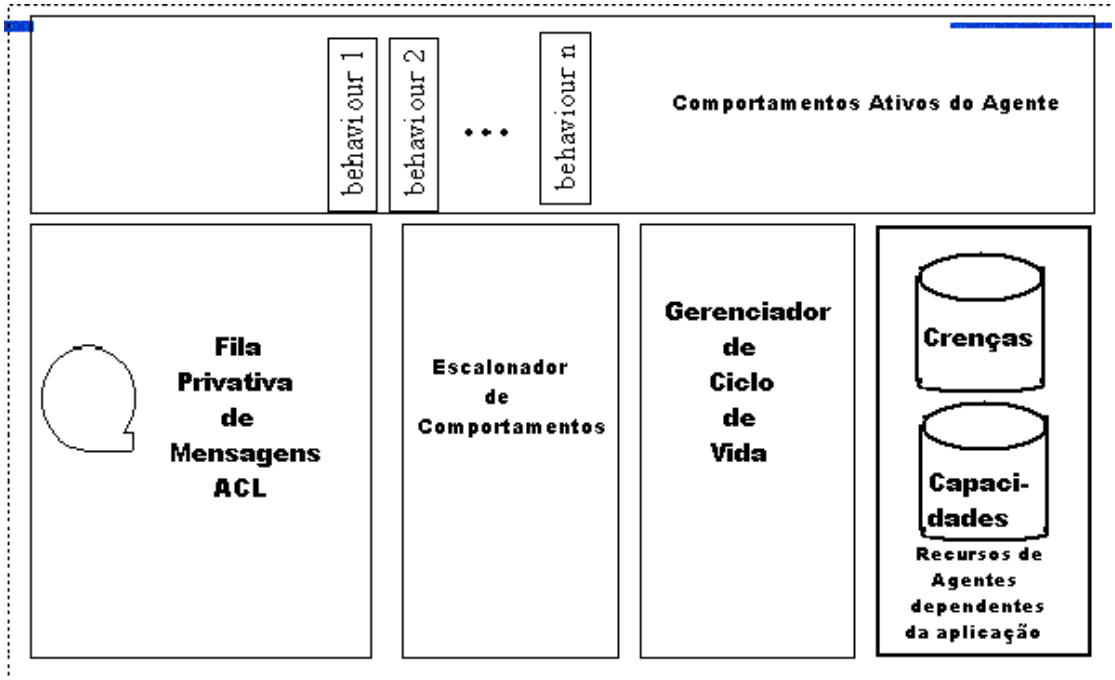

Figura 2.3. Arquitetura interna de uma agente genérico em JADE [JADE 2003]

2.6.1 Ciclo de Vida

Um agente JADE pode está em um dos vários estados de acordo com ciclo de vida (*Agent Platform Life Cycle)* das especificações FIPA. Na Figura 2.4, temos a representação do ciclo de vida de um agente definido pela FIPA e seus estados possíveis.

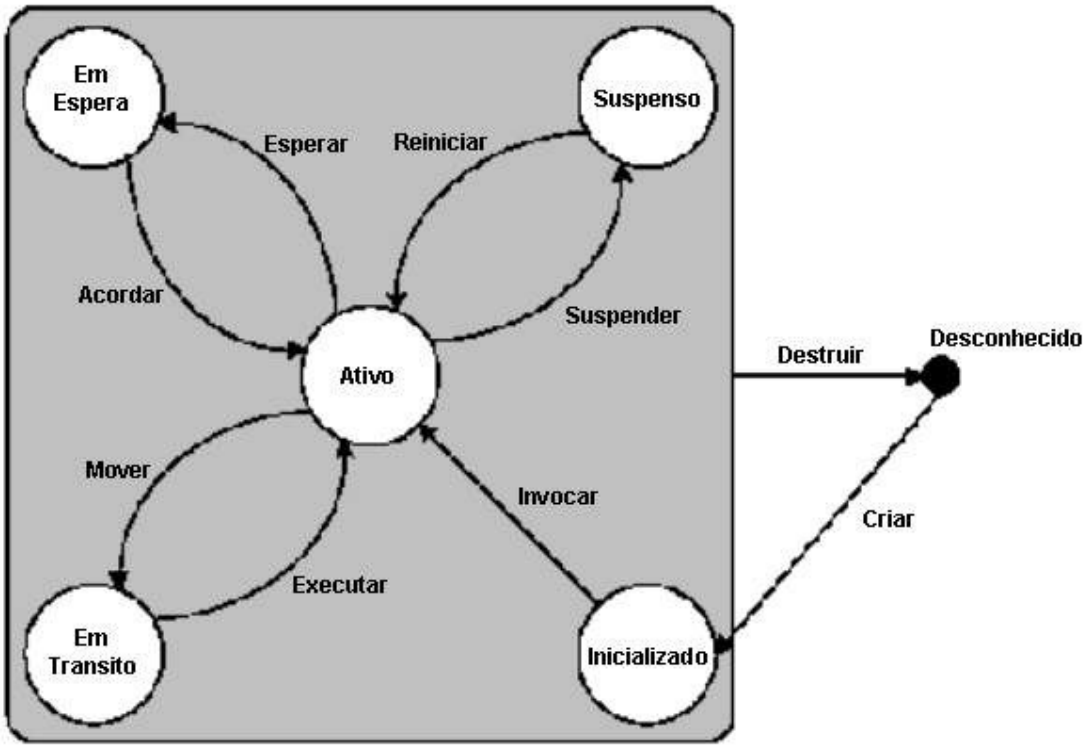

Figura 2.4. Ciclo de Vida de um agente definido pela FIPA [FIPA 2003]

Esses estados são representados em JADE como constantes estáticas da classe *Agent.* São eles:

- **AP_INIATED:** o objeto da classe *Agent* foi instanciado, mas ainda não se registrou no AMS, não tem nome ou endereço e não pode comunicar com outros agentes.
- **AP_ACTIVE:** o objeto da classe *Agent* está registrado no AMS, tem nome formal e endereço e tem acesso a todas funcionalidades do JADE.
- **AP_SUSPENDED:** o objeto da classe *Agent* está no momento interrompido. Sua *thread* interna está suspensa e nenhum comportamento está sendo executado.
- **AP_WAITING:** o objeto da classe *Agent* está bloqueado, esperando por alguma coisa. Sua *thread* interna está "dormindo" (*sleeping)* sob um monitor Java e irá acordar quando alguma condição ocorrer (geralmente quando uma mensagem chega).

- **AP_DELETED:** o agente está definitivamente "deletado" ou encerrado. Sua *thread* interna terminou sua execução e o agente não está mais registrado no AMS.
- **AP_TRANSIT:** um agente móvel entra nesse estado enquanto tiver migrando para uma nova localização. O sistema continua a armazenar mensagens em um *buffer* que serão enviadas para essa nova localização.
- **AP_COPY:** esse estado é usado internamente pelo JADE para agentes que foram clonados.
- **AP_GONE:** esse estado é usando internamente pelo JADE quando um agente móvel migrou para uma outra localização e se encontra em um estado estável.

A troca de estados é feita por meio de métodos como podemos visualizar na Figura 2.4 (reiniciar, esperar, suspender, *etc).*Existem métodos públicos disponibilizados pela classe *Agent* de JADE que fazem as transições entre esses estados acima citados. São eles:

- *public void doActivate ()* – Faz a transição do estado *suspenso* para o *ativo* ou *em espera* (estado que agente estava quando *doSuspend ()* foi chamado). Esse método é chamado da plataforma de agentes e reinicia a execução do agente. Chamar *doActivate ()* em um agente que não esteja suspenso não causará efeito nenhum.

- *public void doSuspend ()* – Faz a transição do estado *ativo* ou *em espera* para o estado de *suspenso.* O estado original do agente é salvo e será restaurado pela chamada ao método *doActivate ()*. Esse método pode ser chamado pela plataforma de agentes ou pelo próprio agente que interrompe todas suas atividades. Mensagens que chegam para um agente suspenso são armazenadas pela plataforma e são entregues assim que o agente reinicie suas atividades. Chamar *doSuspend ()* de um agente suspenso não causa efeito nenhum.

- *public void doWait ()* – Faz a transição do estado *ativo* para o estado *em espera*. Esse método pode ser chamado pela plataforma de agentes ou pelo próprio agente que causa um bloqueio dele mesmo, interrompendo todas suas atividades até que alguns eventos ocorram. Um agente em estado de espera sai desse estado quando método *doWake ()* é chamado ou quando uma mensagem chega. Chamada ao método *doWait ()* em agentes suspensos ou em espera não causa efeito nenhum.

- *public void doWake ()* – Faz a transição do estado *em espera* para o estado *ativo.* Esse método é chamado pela plataforma de agentes e reinicia a execução do agente. Chamada ao método *doWake ()* em agente que não está em espera não tem efeito.

- *public void doDelete ()* – Faz a transição dos estados *ativo, suspenso* ou *em espera* para o estado "*deletado"*. Esse ato causa a destruição do agente. Esse método pode ser chamado pela plataforma de agentes ou pelo próprio agente. Chamar *doDelete ()* em um agente já destruído não causa efeito nenhum.

- *public void doMove (Location* destino*)* – Faz a transição do estado *ativo* para o estado *em transito.* Esse método tem o objetivo de dar suporte à mobilidade de agentes e é chamado pela plataforma de agentes ou pelo próprio agente para iniciar o processo de migração.

- *public void doClone (Location* destino, *java.lang.String* novoNome) – Faz a transição do estado *ativo* para o estado "*copy"*. Esse método tem o objetivo de dar suporte à mobilidade de agentes e é chamado pela plataforma de agentes ou pelo próprio agente para iniciar o processo de clonagem.

Vale ressaltar que é permitido ao agente executar seus comportamentos/*behaviours* apenas quando estiver no estado *ativo.* Outro fato que também merece ser ressaltado é que se em qualquer dos comportamentos de um agente for chamado o método *doWait ()*, o agente como um todo e todas suas atividades serão bloqueadas, não só o comportamento que chamou o método. Ao invés disso, o método *block ()* da classe *Behaviour* evitaria essa bloqueio total, uma vez que permite a suspensão de apenas um comportamento, e além de se desbloquear caso uma mensagem for recebida.

2.6.2 Principais Métodos da Classe *Agent*

Segue abaixo alguns dos principais métodos da classe *Agent*:

- ***protected void setup ()*** – Esse método protegido é indicado para códigos inicializadores da aplicação. Os desenvolvedores podem sobrescrever esse

método fornecendo comportamentos necessários ao agente. Quando este método é chamado, o agente já está registrado no AMS e está apto a enviar e receber mensagens. Porém, o modelo de execução do agente ainda é seqüencial e nenhum escalonamento de comportamento foi efetivado ainda. Logo, é essencial adicionar pelo menos um comportamento para o agente neste método, além de tarefas comuns de inicialização como registro no DF, para torná-lo apto a fazer alguma coisa.

- ***public void addBehaviour (Behaviour* comportamento) -** Esse método adiciona um novo comportamento ao agente. Ele será executado concorrentemente com outros comportamentos. Geralmente é usado no método *setup ()* para disparar algum comportamento inicial, porém pode ser usado também para gerar comportamentos dinamicamente. O método *removeBehaviour ()* faz justamente o inverso: remove determinado comportamento do agente
- ***public final void send (ACLMessage* mensagem)** - Envia uma mensagem ACL para outro agente. Esse agente destino é especificado no campo *receiver* da mensagem no qual um ou mais agentes podem ser definidos como receptores.
- ***public final ACLMessage receive ()*** – Recebe uma mensagem ACL da fila de mensagens do agente. Este método é não-bloqueante e retorna a primeira mensagem da fila caso haja alguma. Retorna *null* caso não haja mensagens.

Existem ainda métodos bastante particulares e úteis que podem ser encontrados na documentação do JADE. Os acima citados são os básicos e primordiais que qualquer sistema multiagente, por menor que seja, tem que ter.

2.7 COMPORTAMENTOS / *BEHAVIOURS*

As ações ou comportamentos que um agente desempenha dentro de um sistema multiagente são fundamentais para que o objetivo final da aplicação seja alcançado. No

JADE, o desenvolvedor que pretende implementar um agente deve obrigatoriamente herdar a classe *Agent* e implementar tarefas específicas para o agente escrevendo um ou mais subclasses *Behaviour,* instanciando-as e adicionando-as ao agente.

JADE contém comportamentos específicos para a maioria das tarefas comuns na programação de agentes, tais como envio e recebimento de mensagens e até construção de tarefas mais complexas (Veja Figura 2.5).

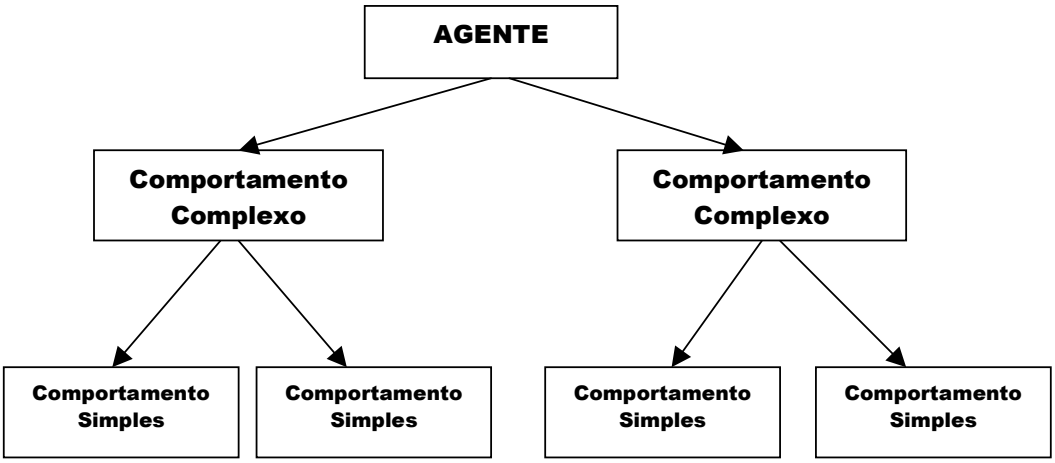

Figura 2.5. Comportamentos compostos por sub-comportamentos em JADE

A estrutura de comportamentos do JADE dá-se através de um escalonador, interno à super classe *Agent*, que gerencia automaticamente o escalonamento desses comportamentos (*behaviours).* Do ponto de vista de programação concorrente um agente é um objeto ativo com uma *thread* de controle interna. JADE usa o modelo de uma *thread* por agente ao invés de uma *thread* por tarefa ou conversação, com o objetivo de manter um número pequeno de *threads* necessárias para executar a plataforma de agentes. Caso criasse uma nova *thread* para cada comportamento, o desempenho do sistema poderia ser comprometido.

O escalonamento desses comportamentos é feito por um escalonador, implementado na super classe *Agent* e abstraído ao programador, que realiza a política de *round-robin*[10] não preemptivo*,* em todos os comportamentos disponíveis na fila de pronto [BELLIFEMINE 2003]. No escalonamento *round-robin* cada comportamento adicionado ao agente é colocado em uma fila.Quando esse comportamento passa para o estado de execução, ou seja, o processo passa para a CPU, existe um tempo limite para a sua utilização. Quando

[10] Mais informações sobre escalonamento de processos e técnicas de escalonamento tais como Round-Robin, First-In-First-Out, etc. podem ser encontradas no livro **Sistemas Operacionais – Conceitos e Aplicações** - autor - Abraham Silberschatz. Editora Campus 1ª edição, 2001.

esse tempo denominado *time-slice* ou quantum, expira, sem que antes o processador seja liberado pelo processo, este volta ao estado de pronto, dando a vez para outro processo. Isso permite a execução da classe de um comportamento completo. Ou seja, se a tarefa ou comportamento cede sem o controle ter sido completado, ela será re-escalonada no próximo ciclo. Ela poderá não ser re-escalonada caso esteja bloqueada (um *behaviour* pode se bloquear no instante em que espera mensagens para evitar desperdício de tempo de CPU e evitar uma "espera ocupada").

Em outras palavras, um comportamento é executado por vez por um determinado tempo (quantum). Quando acaba esse tempo, o comportamento pode voltar para o fim da fila, caso não tenha acabado de executar sua tarefa.

O fato de ser não preemptivo significa dizer que não existe prioridade entre os comportamentos, ou seja, cada comportamento adicionado deve ir para o fim da fila e todos têm a mesma fatia de tempo.

2.7.1 Classe Behaviour

Behaviour é uma classe abstrata do JADE disponível no pacote *jade.core.behaviours.* Uma classe abstrata é uma classe que possui alguns métodos implementados e outros não, e não pode ser instanciada diretamente. Ela tem como finalidade provê a estrutura de comportamentos para os agentes JADE implementarem.

O principal método da classe *Behaviour* é o *action ()*. É nele que serão implementadas as tarefas ou ações que este comportamento irá tomar. Outro importante método é o *done ()*, que é usado para informar ao escalonar do agente se o comportamento foi terminado ou não. Ele retorna *true* caso o comportamento tenha terminado e assim este é removido da fila de comportamentos, e retorna *false* quando o comportamento não tenha terminado obrigando o método *action ()* a ser executado novamente.

Outros importantes atributos e métodos da classe *Behaviour* são:

- *protected Agent myAgent* – Retorna o agente a qual esse comportamento pertence.

- *void block ()* – Bloqueia esse comportamento. Pode ser passado como parâmetro um o tempo de bloqueio em milisegundos: *void block (long* tempo).
- *void restart ()* – Reinicia um comportamento bloqueado.
- *void reset ()* – Restaura o estado inicial do comportamento.
- *boolean isRunnable ()* – Retorna se o comportamento está bloqueado ou não.

2.7.2 Classes Derivadas do *Behaviour*

Como já foi mencionado antes, JADE possui várias classes de comportamentos prontas para uso pelo desenvolvedor adequando-as de acordo com a necessidade específica do agente. Hierarquicamente, eles estão estruturados da seguinte forma:

⇒ Classe *jade.core.behaviours.**Behaviour***
- ⇒ Classe *jade.core.behaviours.**CompositeBehaviour***
  - ⇒ Classe *jade.core.behaviours.**FSMBehaviour***
  - ⇒ Classe *jade.core.behaviours.**ParallelBehaviour***
  - ⇒ Classe *jade.core.behaviours.**SequentialBehaviour***
- ⇒ Classe *jade.core.behaviours.**ReceiverBehaviour***
- ⇒ Classe *jade.core.behaviours.**SimpleBehaviour***
  - ⇒ Classe *jade.core.behaviours.**CyclicBehaviour***
  - ⇒ Classe *jade.core.behaviours.**OneShotBehaviour***
    - ⇒ Classe *jade.core.behaviours.**SenderBehaviour***
  - ⇒ Classe *jade.core.behaviours.**TickerBehaviour***
  - ⇒ Classe *jade.core.behaviours.**WakerBehaviour***

Como classes mais gerais temos a *CompositeBehaviour, ReceiverBehaviour e a SimpleBehaviour.* Sendo que, algumas destas são subdivididas em outras classes para tornar mais específicas suas finalidades.

Na Figura 2.6 temos um diagrama de classes em UML que mostra a hierarquia de classes que derivam da classe abstrata *Behaviour.* As classes *CompositeBehaviour* e *SimpleBehavior* são mostradas como classes derivadas da *Behaviour* (a classe

ReceiverBehaviour foi omitida no diagrama por não ter classes filhas). Também é possível visualizar as classes que herdam de *CompositeBehaviour* e *SimpleBehaviour* e um pequeno resumo do tipo de tarefas que elas modelam.

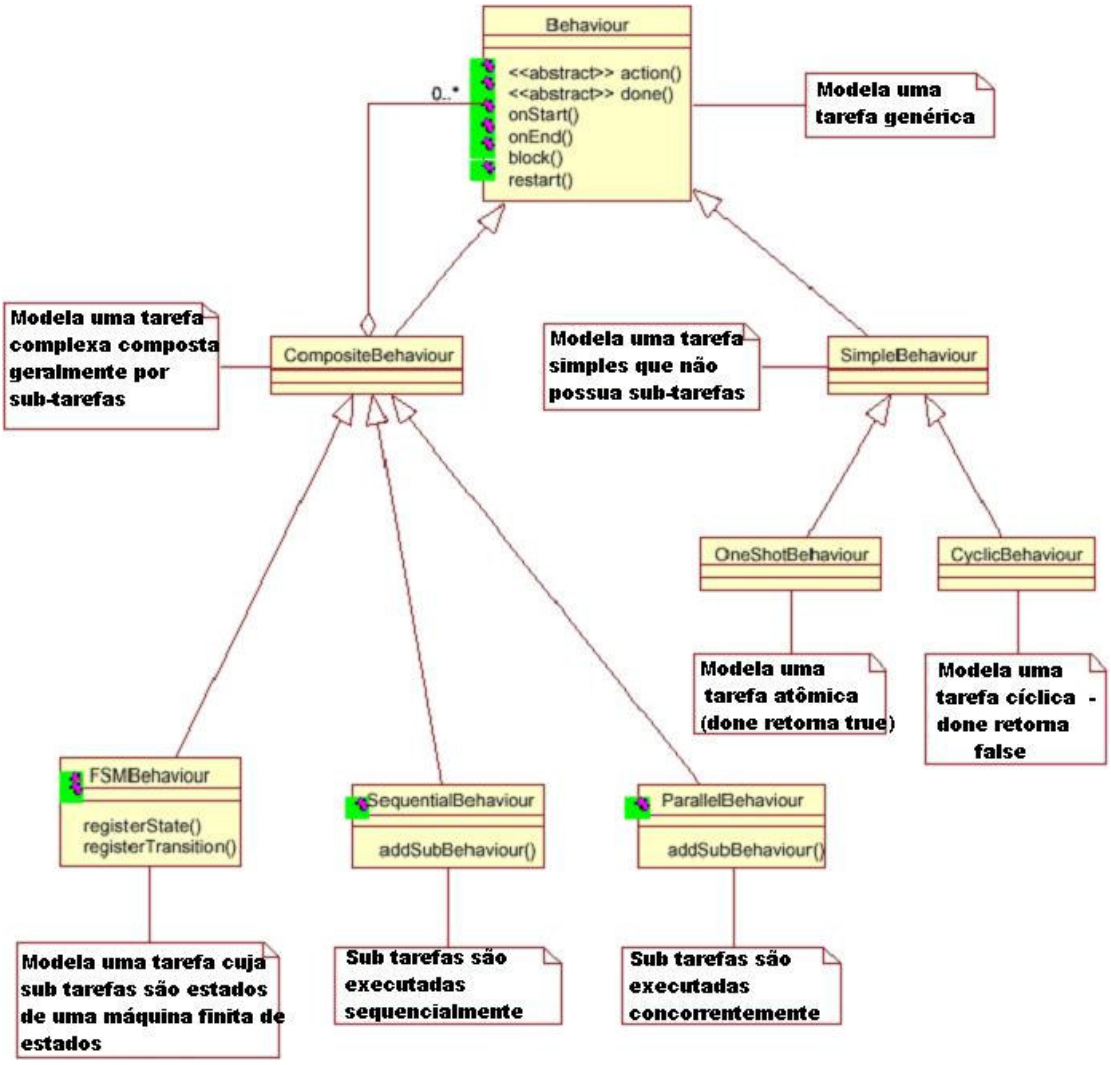

Figura 2.6. Hierarquia das classes derivadas de *Behaviour* em UML

Falaremos agora de todas as classes acima citadas com mais detalhes e outras classes de comportamentos que também acompanham no JADE.

A classe *ReceiverBehaviour* implementa um comportamento para recebimento de mensagens ACL. Ela encapsula o método *receive ()* como uma operação atômica[11], tendo o comportamento encerrado quando uma mensagem ACL é recebida. Possui o método *getMessage ()* que permite receber a mensagem.

[11] Operação atômica é uma operação que se realiza inteiramente sem nenhuma interrupção.

Já a classe *CompositeBehaviour* é uma classe abstrata para comportamentos, como o próprio nome diz, compostos por diferentes partes. Ela controla internamente comportamentos filhos dividindo seu "quantum" de execução de acordo com alguma política de escalonamento. Possui três classes herdadas disponibilizadas pelo JADE. São elas:

- *jade.core.behaviours.**FSMBehaviour*** – Comportamento composto baseado no escalonamento por máquina de estados finitos (***F**inite **S**tate **M**achine*). O *FSMBehaviour* executa cada comportamento filho de acordo com uma máquina de estados finitos definido pelo usuário. Mais especificamente cada comportamento-filho representa um estado na máquina de estados finitos. Ela fornece métodos para registrar estados (sub-comportamentos) e transições que definem como dar-se-á o escalonamento desses comportamentos-filho. Os passos básicos para se definir um *FSMBehaviour* são:
  1) Registrar um comportamento único como estado inicial através do método *registerFirstState* passando como parâmetros o *Behaviour* e o nome do estado.
  2) Registrar um ou mais comportamentos como estados finais através do método *registerLastState*
  3) Registrar um ou mais comportamentos como estados intermediários através do método *registerState*
  4) Para cada estado, registrar as transições deste com os outros estados através do método *registerTransition.*
- *jade.core.behaviours.**ParallelBehaviour*** - Comportamento composto com escalonamento concorrente dos comportamentos filhos. *ParallelBehaviour* executa seus comportamentos filhos concorrentemente e finaliza quando uma condição particular em seus sub-comportamentos é atingida. Por exemplo, quando um número "X" de comportamentos-filho terminarem ou um comportamento-filho qualquer terminar ou quando todos os comportamentos-filho terminarem. Essas condições são definidas no construtor da classe, passando como parâmetro as constantes ***WHEN_ALL***, quando for todos, ***WHEN_ANY***, quando for algum, ou um valor inteiro que especifica o número de comportamentos filhos terminados que são necessários para finalizar o *ParallelBehaviour.* O método para adicionar comportamentos-filho é o *addSubBehaviour.*

- *jade.core.behaviours.**SequentialBehaviour*** - Comportamento composto com escalonamento seqüencial de comportamentos-filho. Ou seja, os comportamentos-filho são executados numa ordem seqüencial e só é encerrado quando o ultimo comportamento-filho é finalizado. O método para adicionar comportamentos-filho é o *addSubBehaviour.*

Por último, temos a classe abstrata *SimpleBehaviour.* Ela é um comportamento atômico. Isto é, modela comportamentos que são feitos para serem únicos, monolíticos e que não podem ser interrompidos. Possui quatro classes herdadas disponibilizadas pelo JADE. São elas:

- *jade.core.behaviours.**CyclicBehaviour*** – Comportamento atômico que deve ser executado sempre. Essa classe abstrata pode ser herdada para criação de comportamentos que se manterão executando continuamente. No caso, o método *done ()* herdado da super classe *Behaviour,* sempre retorna *false*, o que faz com que o comportamento se repita como se estivesse em um "loop" infinito.
- *jade.core.behaviours.**TickerBehaviour*** – Comportamento executado periodicamente tarefas específicas. Ou seja, é uma classe abstrata que implementa um comportamento que executa periodicamente um pedaço de código definido pelo usuário em uma determinada freqüência de repetições. No caso, o desenvolvedor redefine o método *onTick* () e inclui o pedaço de código que deve ser executado periodicamente. O período de "ticks" é definido no construtor da classe em milisegundos.
- *jade.core.behaviours.**WakerBehaviour -*** Comportamento que é executado depois de determinado tempo expirado. Em outras palavras, é uma classe abstrata que uma tarefa "*OneShot*" que é executada apenas depois que determinado tempo é expirado. No caso, a tarefa é inserida no método *handleElapsedTimeout* () o qual é chamado sempre que o intervalo de tempo é transcorrido.
- *jade.core.behaviours.**OneShotBehaviour*** – Comportamento atômico que é executado uma única vez. Essa classe abstrata pode ser herdada para a criação de comportamentos para tarefas que precisam ser executadas em apenas uma única vez. Tem como classe filha a classe *SenderBehaviour.*

- *jade.core.behaviours.**SenderBehaviour*** – é um comportamento do tipo *OneShotBehaviour* para envio de mensagens ACL. Essa classe encapsula o método *send* () como uma operação atômica. No caso, esse comportamento envia determinada mensagem ACL e se finaliza. A mensagem ACL é passada no construtor da classe.

2.7.3 Particularidade

Devido a modelo não-preemptivo de escolha de comportamentos dos agentes, os programadores de agentes devem evitar uso de "*loops*" infinitos e até executar atividades muito longas dentro do método *action* () dos *Behaviours*. Isso porque enquanto o método *action* de algum comportamento estiver sendo executado, nenhum outro comportamento deste mesmo agente poderá ser executado até que o fim do método *action* ocorra.

Além disso, como não há armazenamento em pilha, o método *action* () sempre será executado do início. Não sendo possível, por exemplo, interromper um comportamento no meio de um *action* (), passar o controle para outros comportamentos e voltar para o comportamento original de local onde tinha sido interrompido [BELLIFEMINE 2003].

2.8 TROCA DE MENSAGENS

Toda a troca de mensagens realizada no JADE é feita através de métodos próprios e com o uso de instâncias da classe *ACLMessage.* Esta classe possui um conjunto de atributos que estão em conformidade com as especificações da FIPA, implementando a linguagem FIPA-ACL. Assim, um agente que pretenda enviar uma mensagem deve instanciar um objeto da classe *ACLMessage*, preenchê-los com as informações necessárias e chamar o método *send* () da classe *Agent*. Caso for receber mensagens, o método *receive ()* ou *blockingReceive* () da

classe *Agent* deve ser chamado. Outra meio de enviar ou receber mensagens no JADE é através do uso das classes de comportamentos *SenderBehaviour* e *ReceiveBehaviour*. Fato que torna possível que as trocas de mensagens possam ser escalonadas como atividades independentes de um agente.

A classe *ACLMessage* implementa, como o próprio nome diz, uma mensagem na linguagem FIPA-ACL complacente às especificações da FIPA. Todos seus atributos podem ser acessados via métodos *set* e *get* (Por exemplo, *getContent* e *setContent)*. Além disso, a *ACLMessage* define um conjunto de constantes (ACCEPT_PROPOSAL, AGREE, CANCEL, CFP,CONFIRM**,** etc) que são usadas para se referir às performativas da FIPA. Essas constantes são referidas no construtor da classe com o objetivo de definir o tipo de mensagem.

Os métodos da classe *ACLMessage* são simples e abrangentes. Segue abaixo os principais:

- *public void* ***addReceiver*** ***(AID*** **idAgente)** – Adiciona o AID de um agente como receptor ou destinatário da mensagem. Em outras palavras, determina quem receberá a mensagem
- *public void* ***removeReceiver*** **(AID idAgente)** – Remove o AID do agente da lista de receptores.
- *public ACLMessage* ***createReply*** **()**- Cria uma nova mensagem ACL de resposta à determinada mensagem. Assim, o programador só necessita definir o ato comunicativo (*communicate-act)* e o conteúdo da mensagem.
- *public static java.lang.String[]* ***getAllPerformativeNames*** **()** - Retorna um array de strings com todas as performativas.
- *public Iterator* ***getAllReceiver*** **()** – retorna um *iterator* contendo todos os AIDs dos agentes receptores da mensagem.
- *public java.lang.String* ***getContent*** **()** - Retorna uma string contendo o conteúdo da mensagem.
- *public void* ***setContent*** ***(java.lang.String*** **conteúdo)** – Define o conteúdo da mensagem a ser enviada.
- *public void* ***setOntology*** ***(java.lang.String*** **ontologia)** – Define a ontologia da mensagem.

Por padrão FIPA, todas trocas de mensagens entre agentes devem se basear em *strings* ou textos e é o que ocorre em JADE no qual os métodos *getContent* e *setContent* trabalham com *strings*. Porém JADE também permite que não só *strings* sejam transmitidas por mensagens, mas também outros objetos Java. No método *setContentObject* da classe *AclMessage* permite definir como conteúdo de uma mensagem um objeto Java do tipo *java.io.Serializable*. Já o método *getContentObject* retorna o conteúdo definido pelo método anterior. Nas Figuras 2.7 e 2.8 abaixo seguem exemplos do uso desse dois métodos.

```
ACLMessage msg = new ACLMessage (ACLMessage.INFORM) ;
Date data = new Date () ;
try{
  msg.setContentObject (data);
} catch (IOException e) {}
```

Figura 2.7 Exemplo do uso de setContentObject

```
ACLMessage msg = receive ();
try {
  Date data =  (Date) msg.getContentObject();
}
catch (UnreadableException e) {}
```

Figura 2.8 Exemplo do uso de getContentObject

Porém o próprio JADE não incentiva o uso desses métodos por estes não estarem em conformidade às especificações da FIPA.

2.9 INTEROPERABILIDADE

Conforme já foi mencionado antes, a arquitetura de JADE é baseada em *Java Virtual Machine* em diferentes *hosts*. Para manter as comunicações entre diferentes ambientes

JADE utiliza-se do protocolo IIOP[12] (*Internet Inter-ORB Protocoll*). Para comunicação de agentes na mesma plataforma JADE já utiliza outros meios (RMI ou via eventos). Ou seja, podemos perceber que JADE possui um comportamento variado em relação a forma com que realiza sua comunicação, aonde o mecanismo é selecionado de acordo com a situação do ambiente. Isso ocorre com um objetivo único de atingir o menor custo possível de passagens de mensagens. No caso, quando um agente JADE envia uma mensagem, as seguintes situações são possíveis:

- Se o agente receptor reside no mesmo container de agentes, então o objeto Java representante da mensagem ACL é passado para o receptor via eventos. Sem invocações remotas.
- Se o agente receptor reside na mesma plataforma JADE, mas em containeres diferentes, a mensagem ACL é enviada usando Java RMI.
- Se o agente receptor reside em uma diferente plataforma de agentes, o protocolo IIOP é usado de acordo com o padrão FIPA.

Isso é ilustrado na Figura 2.9, o qual mostra as possibilidades que JADE tem para realizar suas trocas de mensagens. O meio mais eficiente é escolhido de acordo com a localização do agente receptor da mensagem conforme foi explicado anteriormente. Note que na parte inferior da Figura 2.9 existem as formas possíveis de comunicação. Esse cache local, localizado no ACC (parte central da Figura 2.9), armazena as referências dos objetos dos outros containeres. Essas referências são adicionadas ao cache sempre que uma mensagem é enviada.

[12] Introduzido na segunda especificação de CORBA (*Common Object Request Broker Architecture*), o IIOP permitiu que CORBA se tornasse uma solução definitiva para interoperabilidade entre objetos que não estão presos a uma plataforma ou padrão específico. Mais informações sobre CORBA e IIOP podem ser encontradas no livro **Distributed Systems – Concepts and Design** de Tim Kindberg, George Coulouris & Jean Dollimore, 3ª edição, editora Addison Wesley Pub 2001.

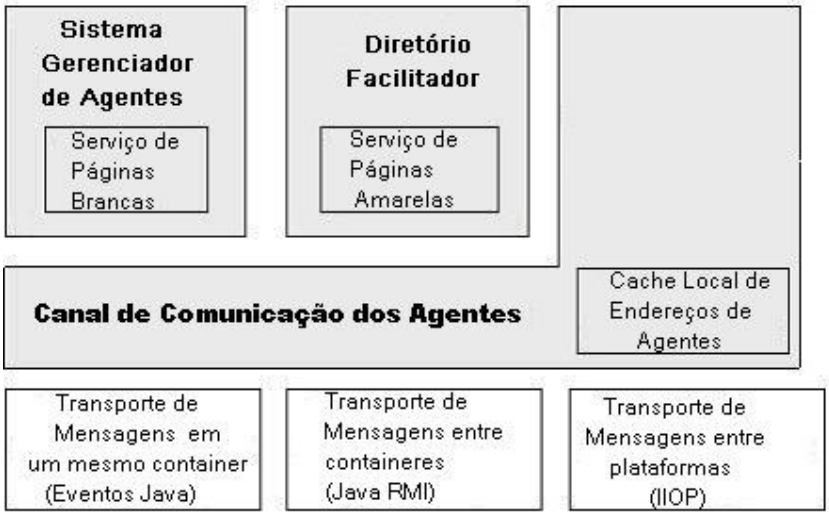

Figura 2.9. Interoperabilidade no JADE

2.10 BIBLIOTECAS DE PACOTES DO JADE

JADE oferece inúmeros pacotes de classes que visam auxiliar tanto na implementação quanto na monitoração de sistemas multiagentes. Os principais pacotes que compõem o JADE são:

- ***Jade.core*** – Esse pacote implementa o "kernel" do sistema. Ou seja, é o "coração" do *Jade* que inclui classes importantes como a classe *Agent e AID.* Ele fornece uma passagem de mensagens simples com protocolo de mensagens variados e um ambiente de execução *multithreading* para softwares de agentes.
- ***Jade.core.behaviours*** – Esse subpacote do pacote *jade.core* contêm a classe *Behaviour.* Essa classe, que próprio nome já diz, implementa os comportamentos básicos de um agente. Ou seja, suas tarefas, intenções ou objetivos. Esses comportamentos foram abordados no tópico *2.7 Comportamentos/Behaviours.*
- ***Jade.lang.acl*** – Neste pacote contém todo o suporte necessário para a implementação de uma mensagem no formato FIPA-ACL , incluindo a classe

ACLMessage, o analisador (*parser*), o codificador (*encoder*), e uma classe para representar *templates* (padrões) de mensagens ACL. Esse assunto foi abordado no tópico *2.8 Troca de Mensagens.*

- ***Jade.content*** – Este pacote contém um conteúdo de classes para suportar as ontologias definidas pelo usuário e as linguagens de conteúdo.
- ***Jade.domain*** - Este pacote e seus sub-pacotes contêm todas as classes Java que representam as entidades gerenciadoras de agentes (*Agent Management)* definidas pela FIPA. Tais como:
    - *AMS Agent:* gerencia o ciclo de vida dos agentes no ambiente e guarda informações sobre eles.
    - *DF Agent:* mantêm as páginas amarelas (*yellow-pages)* de informações sobre os agentes.
- ***Jade.gui*** – Este pacote contém componentes que podem ser usados para construir interfaces gráficas Swing[13] genéricas para mostrar e editar propriedades relacionadas à agentes JADE como *Agent Identifiers (AIDs),* descrições de agentes, *ACLMessages, etc.*
- ***Jade.mtp*** **–** Este pacote contém todas as interfaces Java que todo protocolo de transporte de mensagens (*Message Transport Protocol* - MTP) deve implementar para que fique totalmente integrada com o *framework* do JADE como também a implementação desses protocolos.
- ***Jade.proto*** – Este contém classes para modelar protocolos de interação padronizados (Ex: *fipa-request, fipa-query, fipa-contract-net, fipa-subscribe,* e outros definidos pela FIPA), como também classes abstratas para que os programadores desenvolvam seus próprios protocolos. Para cada protocolo de interação, de acordo com as especificações FIPA, dois papéis (*roles*) podem ser desempenhados por um agente:
    - *Initiator:* O agente contacta um ou mais agentes para iniciar uma conversação de acordo com um específico protocolo de interação.

[13] Swing é uma biblioteca de interface de usuário (GUI) desenvolvida pela Sun Microsystems para a linguagem Java e presente a partir do Java 2.

- *Responder:* Em resposta a uma mensagem recebida de outro agente, o agente continua em uma nova conversação seguindo um protocolo específico.

- ***Jade.wrapper*** - Juntamente com o *jade.core.Profile* e o *jade.core.Runtime* este pacote contém todo o suporte necessário para que aplicações Java externas usem JADE como um tipo de biblioteca e executem o "run-time" do JADE, agentes JADE e os containeres de agentes pela própria aplicação.

- ***Jade.util*** – Este pacote contém classes úteis, e em particular classes para tratamento de propriedades e o sub-pacote *leap* que é uma substituição para o *framework* Java que não foi suportada pelo J2ME[14].

- ***Jade.tools*** – Pacote que possui ferramentas úteis para a administração e desenvolvimento dos agentes JADE.

2.11 OUTRAS CLASSES DE JADE

JADE possui uma grande variedade de classes prontas para ser usadas pelos desenvolvedores de sistemas multiagentes. Por ter código aberto, inúmeras novas classes *add-ons* para JADE estão sendo desenvolvidas e disponibilizadas gratuitamente. Sendo assim impossível falar de todas as classes que JADE possui, porém falaremos de apenas algumas delas visando demonstrar outras funcionalidades que essa plataforma possui.

2.11.1 Classe AID

Localizada no pacote *jade.core.AID*, a classe AID representa um identificador do agente JADE. Esse identificador é único e é utilizado por tabelas internas do JADE para armazenar nomes e endereços dos agentes e também como referência para que os outros

[14] Java 2 Micro Edition ou J2ME é uma versão da linguagem Java para dispositivos móveis (Ex: palmtops). Mais informações no site http://java.sun.com/j2me/

agentes possam trocar mensagens. Geralmente composto por um nome *string* e um endereço composto por uma porta (a *default* é 1099) e o *host* da máquina em que o agente reside.

2.11.2 Classe MessageTemplate

A classe *MessageTemplate* permite construir padrões para se determinadas mensagens estão compatíveis com os padrões definidos. Usando os métodos desta classe, o programador poderá criar um ou mais padrões para cada atributo da mensagem ACL. Além disso, padrões básicos podem ser combinados com operadores AND, OR ou NOT com o objetivo de construir regras de validações mais complexas.

2.11.3 Classe DFService

Essa classe provê um conjunto de métodos estáticos para a comunicação com um serviço de DF complacente com as especificações da FIPA. Inclui métodos para registrar, cancelar registros, alterar e pesquisar em um DF. Está presente no pacote *jade.domain.*

2.11.4 SocketProxyAgent

Essa classe provê um agente JADE que pode ser usado para comunicação com clientes remotos via *sockets*. É uma ferramenta útil que pode por exemplo provê interações com Java *Applets* em *browsers* ou tratar conexões com *firewalls*.

2.11.5 Pacote jade.gui

O pacote *jade.gui* possui inúmeras classes que podem ser usadas para construção de aplicações Java Swing para agentes. Como classes principais têm:

- ***jade.gui.GuiAgent*** - Essa classe abstrata, que é uma extensão da classe *jade.core.Agent*, tem como objetivo gerenciar as *threads* de agentes ativos em aplicações que utilizem uma GUI. Isso é necessário porque quando o programa instancia uma GUI, o Java inicia uma nova *thread* que, logicamente, é

diferente da *thread* agente. Essa fica ativa por causa da *thread* agente uma vez que os comportamentos do agente geralmente vão depender de ações do usuário (Ex: Pressionando um botão, movendo mouse, etc). Portanto, um mecanismo apropriado para gerenciar a interação entre essas duas *threads* ativas se faz necessário. É aí o que *GuiAgent* entra. O que ele faz é definir um espaço de execução para cada *thread* ao invés de uma *thread* ficar requisitando a outra para executar métodos. Esse controle é feito através de eventos (*GuiEvents)* em métodos que a própria classe disponibiliza (onGuiEvent *()* e *postGuiEvent ()).* Em suma, seria um agente para GUIs que controla agentes JADE.

- ***jade.gui.GuiEvent*** - Essa classe define um objeto do tipo *GuiEvent* que é usado para notificar um evento para um *GuiAgent*. Possui dois atributos obrigatórios: a origem do evento e um inteiro identificando o tipo de evento.

- ***jade.gui.AclGui*** – A *AclGui* é uma classe derivada de JPanel do pacote Swing do Java com a adição de todos os controles necessários para editar e visualizar corretamente os campos de uma mensagem ACL. Existem basicamente duas maneiras de usar a *AclGui*:

    - Modo Não Estático – A *AclGui* funciona como uma JPanel comum, sendo adicionada ao Container de um objeto Java. Os métodos *setMsg ()* e *getMsg ()* podem ser usados para mostrar/editar determinada mensagem no *panel*. Os métodos *setEnabled ()* e *setSenderEnabled ()* podem ser usados para habilitar e desabilitar modificações nos campos da mensagem ou no campo do remetente.
    - Modo Estático – No caso, ela provê métodos estáticos que disparam uma janela de dialogo temporária, que inclui o painel *AclGui* mais botões de OK e CANCEL, permitindo a visualização e edição de determinadas mensagens. Esses métodos são *editMsgInDialog ()* e *showMsgInDialog ()*.

- ***jade.gui.AIDGui*** - É uma classe derivada da *javax.swing.JDialog* que mostra em uma gui o AID de um agente. O método destinado a esse fim é o *ShowAIDGui ().*

Além dessas citadas acima, JADE dispõe ainda nesse pacote mais de 20 classes para utilização relacionadas com Gui's com as mais variadas utilidades, tais como *JadeLogoButton, AboutJadeAction, AgentTree, BrowserLauncher, VisualAIDList* entre outras.

3. JADE – PARTE TÉCNICA

3.1 INSTALAÇÃO

3.1.1 Requisitos Mínimos

Por ser baseado em Java, o requisito mínimo para executar a plataforma JADE é o *run-time* do Java. No caso, *Java Run-Time Environment* (JRE) versão 1.2 ou superior. Já para o desenvolvimento usando o *framework* do JADE é necessário no mínimo o *Java Development*[15] *Kit (JDK)* 1.2 ou superior. Em termos de hardware a configuração mínima é um Pentium II com 20 Megabytes de disco rígido e 128 megabytes de memória RAM.

3.1.2 Software JADE

Todo o software do JADE, inclusive *add-ons* e novas versões, estão distribuídos, sob as limitações[16] da LGPL, e disponíveis para *download* no *web site* do JADE –

[15] As versões mais atualizadas do Java e as suas mais variadas plataformas podem ser encontradas e disponíveis para *download* gratuitamente no site http://java.sun.com

[16] A licença do LGPL fornece alguns direitos e também deveres. Dentre os direitos estão o acesso ao código fonte do software, a permissão de fazer e distribuir cópias, permissão para fazer melhorias e funcionalidades e incorporar o JADE a programas proprietários. Já os deveres são: jamais fazer modificações privadas e secretas e não alterar a licença do JADE e suas modificações.

http://jade.cselt.it . No caso, para a versão 3.0b1, cinco arquivos comprimidos são disponibilizados:

1. **jadeSrc.zip** (1.4 Megabytes) - código fonte do JADE
2. **jadeExamples.zip** (227 Kilobytes) - código fonte de exemplos de programas feitos em JADE.
3. **jadeDoc.zip** (3.2 Megabytes) - Toda a documentação, incluindo o *javadoc* da API do JADE e manuais de referência.
4. **jadeBin.zip** (1.2 Megabytes) - O JADE binário pronto para uso.
5. **jadeAll.zip** (5.8 Megabytes) - Todos os arquivos acima citados agrupados.

Como sugestão de teste, [JADE 2003] sugere que após a instalação as seguintes linhas de comando sejam digitadas com a finalidade de averiguar se JADE está corretamente instalado:

- java -classpath lib\jade.jar; lib\jadeTools.jar; lib\Base64.jar; lib\iiop.jar jade.Boot -gui
- java -jar lib\jade.jar -gui -nomtp (Equivalente à primeira)
- demo/rundemo.bat (Executa um exemplo)

3.2 EXECUÇÃO DO AMBIENTE JADE

Após a descompressão dos arquivos, por padrão um diretório é criado com o nome de *jade* e subdiretórios: *classes, demo, doc, lib* e *src*. No subdiretório *lib* estão arquivos JAR que terão de ser adicionados na variável de ambiente c*lasspath*.

Depois de configurado o *classpath* é possível a utilização da linha de comando o qual já foi mencionada no tópico anterior: **java jade.Boot [opções] [agentes]**. Com ele é possível executar o *main-container* da plataforma. É nele que os agentes que fazem com que o JADE funcione: DF, AMS, etc.

Containeres adicionais podem ser criados no mesmo *host*. Em *hosts* remotos também é possível através da interligação pelo *main-container* resultando na característica distribuída que JADE possui. Um container adicional de agentes pode ser carregado com a linha de comando: java jade.Boot **-container** [opções] [agentes] .

```
java jade.Boot Option* AgentSpecifier*

Option          = "-container"
                | "-host" HostName
                | "-port" PortNumber
                | "-name" PlatformName
                | "-gui"
                | "-mtp" ClassName "(" Argument* ")"
                    (";" ClassName "(" Argument* ")")*
                | "-nomtp"
                | "-aclcodec" ClassName (";" ClassName )*
                | "-nomobility"
                | "-version"
                | "-help"
                | "-conf" FileName?
                | "-" Keyword Value

ClassName       = PackageName? Word

PackageName     = (Word ".")+

Argument        = Word | Number | String

HostName        = Word ( "." Word )*

PortNumber      = Number

AgentSpecifier = AgentName ":" ClassName ("(" Argument* ")")?

AgentName       = Word

PlatformName    = Word

Keyword         = Word

Value           = Word
```

Figura 3.1. Sintaxe Completa da linha de comando de jade em notação EBNF

Na Figura 3.1 temos a sintaxe em notação EBNF da linha de comando que executa o JADE. Podemos visualizar na figura 3.1 os nomes das opções disponíveis. Abaixo temos todas essas opções com uma pequena descrição de sua funcionalidade:

1 ***-container*** - especifica que essa instância de JADE é um novo container. Por default, essa opção não é selecionada.
2 ***-host*** - especifica o nome do *host* onde o *main-container* será criado ou containeres comuns serão alocados.
3 ***-port*** - especifica a porta onde o *main-container* será criado ou os containeres comuns serão alocados. Por default o número da porta é 1099.
4 ***-name*** - especifica um nome simbólico para ser usado como nome da plataforma. Essa opção só é considerada no caso de um *main-container*.
5 ***-gui*** - especifica que o RMA Gui do JADE deve ser carregado. Por default essa opção não é selecionada.
6 ***-mtp*** - especifica uma lista de MTPs (*Message Transport Protocols)* ou protocolos de transporte de mensagens, para serem ativados nesse container. Por default, o JDK 1.2 IIOP é ativado no *main-container* e nenhum MTP é ativado nos outros containeres.
7 ***-nomtp*** - tem precedência sobre o *-mtp* e o sobrescreve. Ele deve ser usado para sobrescrever o comportamento default do *main-container*

[BELLIFEMINE 2003].

8 ***-aclcodec*** - Por default, todas as mensagens estão codificadas no padrão ACLCodec baseadas em String. Essa opção permite especificar ACLCodec adicionais que se tornarão disponíveis para os agentes codificar/decodificar mensagens.

9 ***-nomobility*** - desabilita a mobilidade e clonagem no determinado container. Desse modo, o container não aceitará requisições de migração de agentes ou clonagem de agentes.

10 ***-version*** - mostra informações de versão do JADE.

11 ***-conf*** - Se nenhum arquivo é especificado após essa opção, a interface gráfica é mostrada que permite carregar/salvar todos os parâmetros de configuração do JADE em um arquivo. Se um arquivo é especificado, todas as opções definidas nesse arquivo são usadas para a execução do JADE.

12 ***-help*** - mostra todas essas informações acima citadas

Além das opções apresentadas acima que se enquadram no argumento [opções], outro argumento opcional disponível é o parâmetro [agentes] que instanciamos agentes na mesma linha de comando do JADE. No caso, uma seqüência de *strings* especifica que agente será instanciado, qual o nome que ele terá e, opcionalmente, quais argumentos serão passados a ele.

Por exemplo, **AgenteJoao:Agentes**, onde **AgenteJoao** é o nome do agente e **Agentes** é classe Java que implementa esse agente. O nome dessa classe pode ser completo, por exemplo, **AgenteJoao:pacotes.agentesjade.Agentes** e será procurado de acordo com a definição no *classpath* do ambiente. Outro exemplo é **AgenteJoao:Agentes ("20 anos" 01)**. No caso, são passados argumentos para a classe **Agentes** que em outras palavras significaria "crie um novo agente chamado *AgenteJoao* o qual será uma instância da classe *Agentes,* passando dois argumentos no seu construtor, 20 anos e 01".

3.2.1 Exemplo

Segue abaixo um pequeno exemplo que mostra a capacidade distribuída de JADE. Antes de tudo, deve-se configurar as variáveis de ambiente, incluindo no *classpath* todos os arquivos JAR que se encontram no subdiretório *lib* do JADE. No caso, segue um exemplo para ser executado em ambientes Windows9.x/NT ou superior.

Primeiramente, configurar o *classpath:*

⇒ Set CLASSPATH = %CLASSPATH%; .; c:\jade\lib\jade.jar; c:\jade\lib\jadeTools.jar;

c:\jade\lib\Base64.jar; c:\jade\lib\iiop.jar;

Supondo que no computador1 tem como *hostname* "PC1", digite a seguinte linha de comando que será executada para carregar o *main-container* :

⇒ Java jade.Boot –gui

Execute a seguinte linha de comando em uma outra máquina a qual criará outro container de agente e fará com que esse container se conecte ao *main-container* no computador1 PC1:

⇒ Java jade.Boot -host PC1 -container Send:examples.receivers.AgentSender

Onde "PC1" é o host da máquina onde se encontra o *main-container, "Send"* é o nome do agente e "examples.receivers.AgentSender" é o código que implementa o agente. Execute agora em uma terceira máquina o seguinte comando que cria dois agentes:

⇒ Java jade.Boot -host PC1 -container Snd:examples.receivers.AgentSender Rcv:examples.receivers.AgentReceiver

Ao final teremos três agentes distribuídos sendo dois do tipo AgentSender e um do tipo AgentReceiver, conectados à plataforma JADE localizada em um *host* diferente dos *hosts* destes agentes. É importante lembrar que as classes destes agentes já deverão está compiladas.

3.3 FERRAMENTAS DE GERENCIAMENTO E DEPURAÇÃO

Conforme, já foi mencionado no capitulo 2, no pacote *jade.tools* encontramos várias classes de ferramentas que JADE disponibiliza para desenvolvedores de agentes. São ferramentas que auxiliam a depuração e o monitoramento de agentes JADE, ajudando na identificação de problemas ou simulação de eventos. Até a versão do Jade 3.0b1[17], as

[17] Até o término desta monografia, a versão mais recente do Jade é a 3.0b1

seguintes ferramentas estão disponíveis no Jade: RMA, DummyAgent, SnifferAgent, e DF GUI.

3.3.1 Remote Management Agent (RMA)

O gerenciador remoto de agentes ou RMA é um console gráfico para o controle e gerenciamento da plataforma JADE. Permite o controle dos estados do ciclo de vida de todos agentes em execução inclusive os distribuídos. Além disso, serve como um controle principal onde estão centralizadas algumas funcionalidades como, por exemplo, chamar as outras ferramentas do JADE.

O RMA é um objeto Java, instanciado da classe *jade.tools.rma.rma.* Pode ser carregado através das seguintes linhas de comando:

- java jade.Boot meuComputador:jade.tools.rma.rma
- java jade.Boot –gui

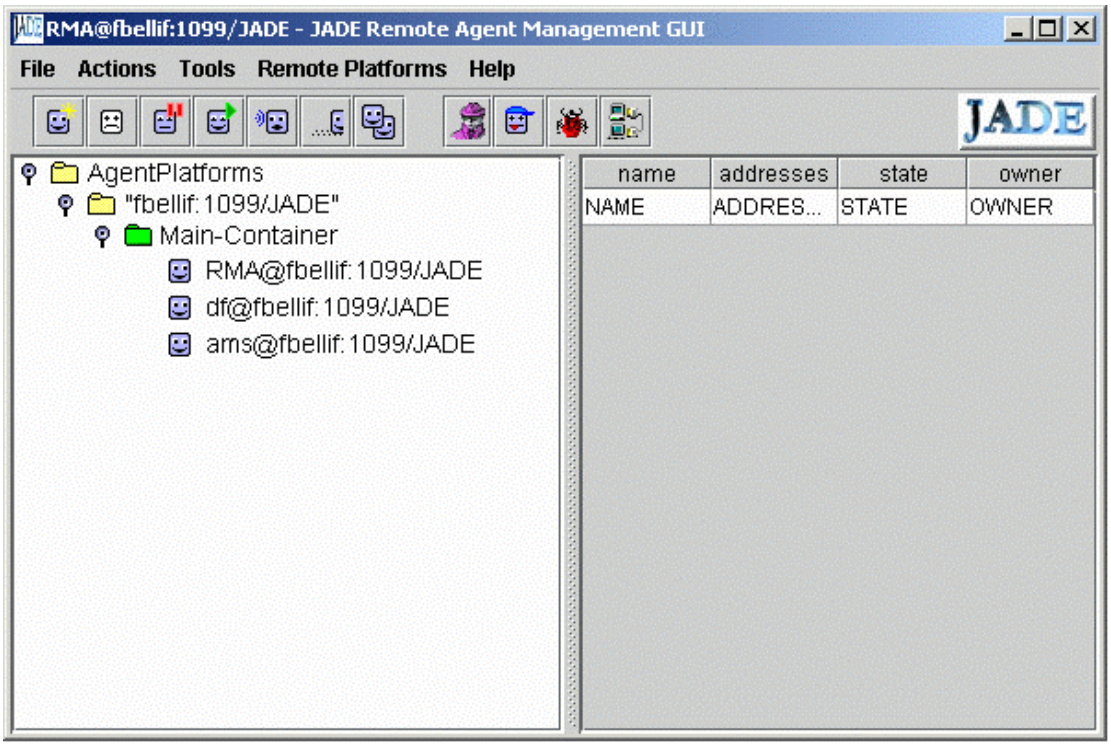

Figura 3.2. Interface Visual do RMA

O RMA oferece o acesso as funcionalidades da plataforma JADE através de um *menu bar*, uma *tool bar* e ícones. Na Figura 3.2, podemos ver a interface do RMA. No lado esquerdo, localiza,-se os containeres em estruturas de árvores aonde a raiz é chamada de "AgentPlatforms". Seu nó filho é um diretório cujo nome é o nome local da plataforma de agentes. Clicando com o botão direito nesse diretório o usuário poderá visualizar a descrição da plataforma de agentes local ("View AP Description"). Outros menus *pop-ups* também estão disponíveis.

Segue abaixo breves explicações sobre as opções de menu disponíveis no RMA:

1. **File Menu** - Contém comandos gerais do RMA. Opções:
    - **Close RMA Agent -** Encerra o agente RMA, chamando o método *doDelete ()* (Lembrando sempre que RMA é um agente como outro qualquer). Vale ressaltar também que isso só afeita o próprio RMA, deixando intacta todo o resto da plataforma de agentes.
    - **Exit this Container** - Encerra o container o qual o RMA está residindo, "matando" o RMA e todos os outros agentes que estão no mesmo processo. Se esse container é o do "front-end" da plataforma de agentes, então toda a plataforma é encerrada.
    - **Shut down Agent Platform** - Como o próprio nome já é claro, finaliza toda a plataforma, encerrando todos os containeres conectados, todos os agentes usuários, e todos os agentes de sistema.
2. **Action Menu -** Esse menu contém itens para invocar as mais variadas ações administrativas necessárias na plataforma como um todo, ou em um conjunto de agentes ou até mesmo em um conjunto de containeres. Opções:
    - **Start New Agent** - Essa ação cria um novo agente, no qual o usuário informa o nome do agente e o nome da classe Java de onde esse novo agente será instanciado. Além disso, se um container está selecionado, o agente é criado e iniciado neste container, senão, o usuário poderá informar o nome do container que ele deseje que o agente inicie.
    - **Kill Selected Items** - Essa ação "mata" todos os agentes e containeres que estão no momento selecionados. "Matar" um agente corresponde a chamar seu método *doDelete ()*, enquanto que "matar" um container corresponde a finalizar todos os agentes que nele residem e cancela o

registro do container da plataforma.

- **Suspend Selected Items** - Essa ação suspende os agentes selecionados chamando o método *doSuspend* () destes. Vale ressaltar que suspender agentes, especialmente o AMS, poderá causar um "*deadlock*" na plataforma.
- **Resume Selected Items** - Essa ação coloca os agentes selecionados de volta no estado AP_ACTIVE, levando em consideração que estes estavam suspensos. Funciona chamando o método *doActivate* ().
- **Change Owner** - Troca o "proprietário" do agente.
- **Send Message** - Envia uma mensagem ACL para um agente.
- **Migrate Agent** - Faz a "migração" de um agente, informando o container para onde o agente irá "mover-se".
- **Clone Agent** - Clona um agente, informando o novo nome do agente clonado e o container aonde ele será iniciado.

3. **Tools Menu -** Esse menu contém comandos para iniciar as outras ferramentas que JADE dispõe.

4. **Remote Platforms Menu -** Esse menu permite controlar algumas plataformas remotas que estão em conformidade com as especificações da FIPA. Lembrando que essas plataformas remotas não necessitam ser obrigatoriamente plataformas JADE. Opções:

   - **Add Platform via AMS AID** - Permite adquirir a descrição (chamada de *APDDescription* na terminologia FIPA) de uma plataforma de agentes remota. O usuário informa o AID do AMS remoto e então a plataforma é adicionada e visualizada na GUI do RMA.
   - **Add Platform via URL** - Mesma função do item anterior, porém ao invés de informar o AID, é informado a URL.
   - **View AP Description** - Para visualizar a descrição da plataforma de agentes selecionada.
   - **Refresh AP Description** - Atualizar a descrição da plataforma de agentes, uma vez que uma plataforma remota pode mudar sua descrição.
   - **Remove Remote Platform** - Remove da GUI a referência da plataforma selecionada.

- **Refresh Agent List** - Essa ação executa uma procura com o AMS da plataforma remota e atualiza a lista de agentes na GUI.

3.3.2 DummyAgent

DummyAgent é uma ferramenta gráfica de monitoramento e "*debugging*" para agentes Jade. Com ela é possível criar e enviar mensagens ACL para outros agentes bem como listar todas as mensagens ACL enviadas e recebidas com informações detalhadas. Além disso é possível salvar essa lista de mensagens em disco e recuperá-la depois. Podem ser abertas muitas instancias de um *DummyAgent.* As duas maneiras de carregar um *Dummy Agent* são:

- Através do RMA GUI, no menu Tools
- Através da linha de comando:

java jade.Boot da:jade.tools.DummyAgent.DummyAgent

Na Figura 3.3 temos a interface visual do Dummy Agent. Do lado direito, é possível visualizar todas as mensagens que o Dummy Agent enviou e também as mensagens que estão recebendo. No lado direito, o usuário pode especificar para qual agente deseja enviar determinada mensagem bem como explicitar que tipo de mensagem, seu conteúdo, ontologias, etc.

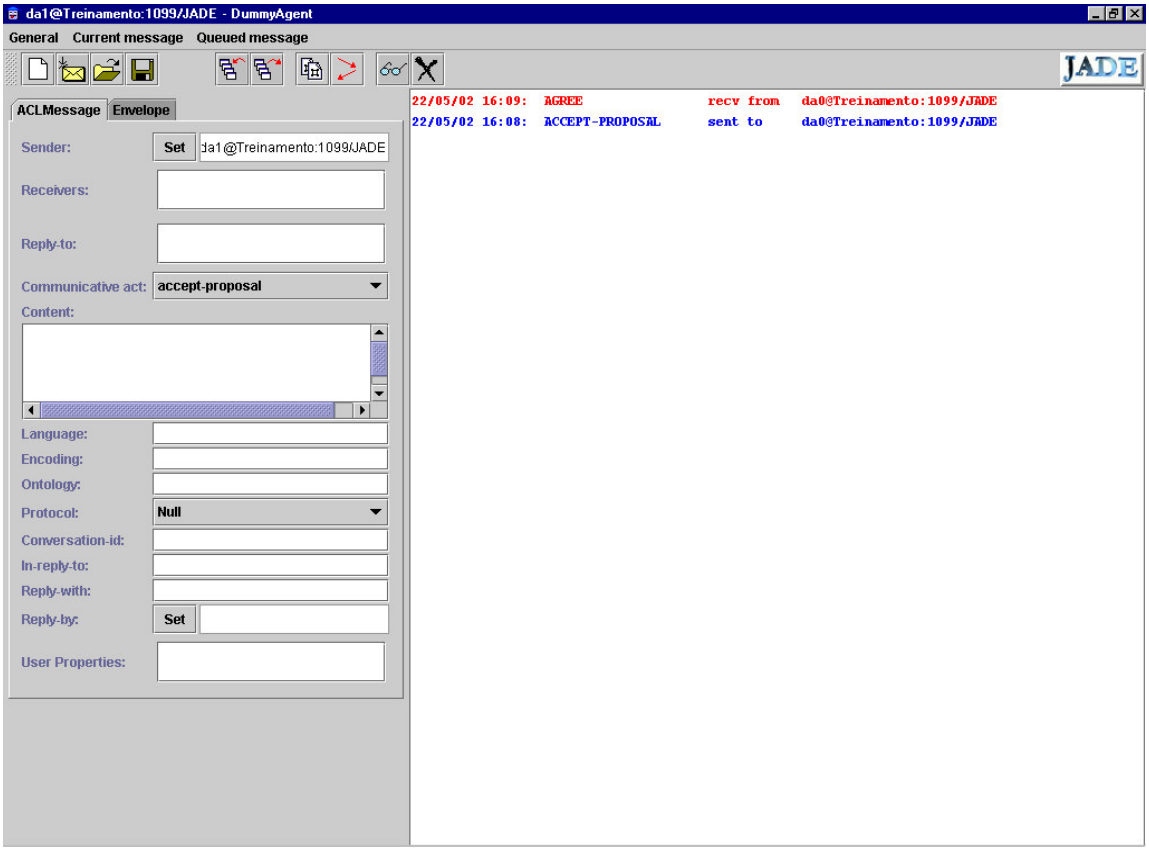

Figura 3.3 Dummy Agent

3.3.3 SnifferAgent

SnifferAgent é uma ferramenta que mostra a troca de mensagens graficamente em notação similar a diagramas de seqüências UML entre determinados agentes. Quando um usuário decide fazer um "*sniff*" em um agente ou grupo de agentes, toda mensagem direcionada a este agente / grupo de agentes ou vinda destes, é rastreada e disponibilizada na GUI. O usuário poderá ver todas mensagens, salvá-las em disco como arquivo-texto ou binário para uso posterior. Ele pode ser iniciado por duas maneiras:

- Através da RMA Gui, no menu Tools
- Através da linha de comando:

java jade.Boot sniffer:jade.tools.sniffer.Sniffer

Na Figura 3.4, temos a interface visual do Sniffer Agent. No lado esquerdo encontram-se todos os agentes registrados na plataforma em um formato bem semelhante ao

RMA. Já no lado direito, encontram-se gráficos que mostram a troca de mensagens entre os agentes especificados pelo usuário. Cada seta representa uma mensagem, sendo possível ter informações detalhadas dessas mensagens clicando nelas.

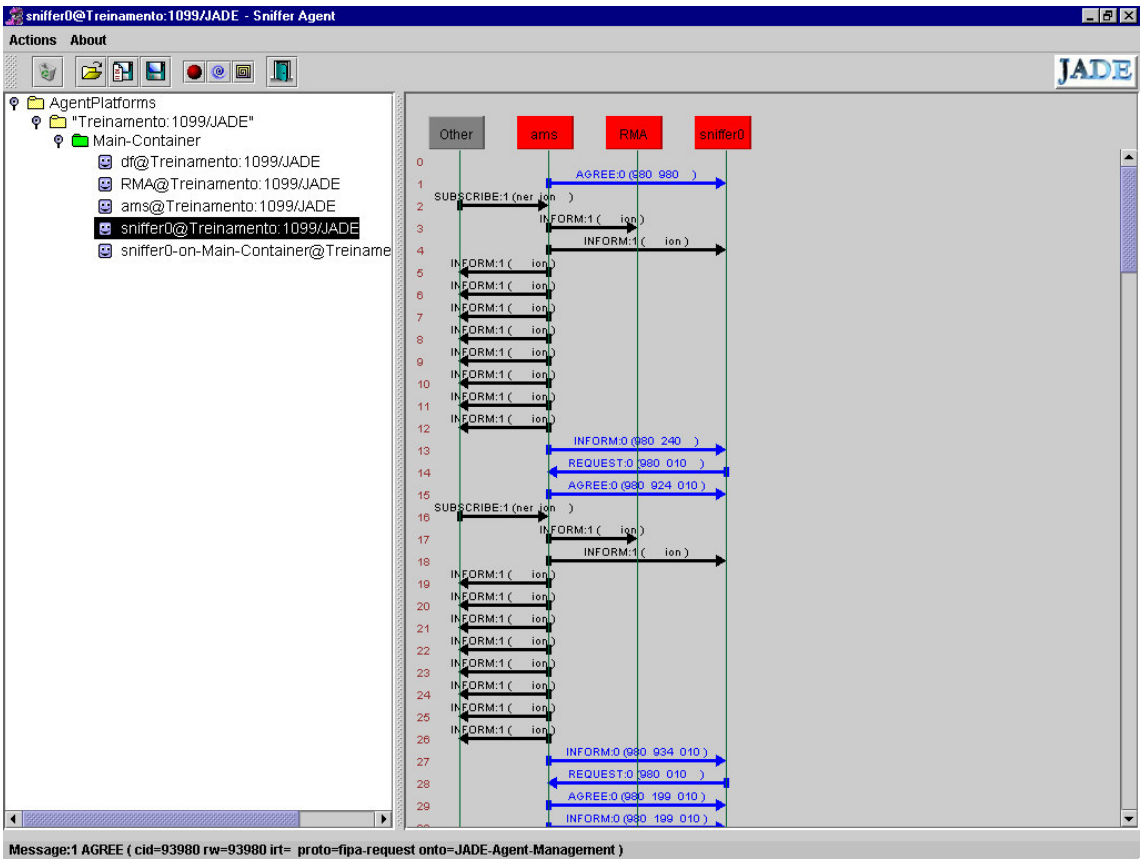

Figura 3.4. Sniffer Agent

3.3.4 Introspector Agent

É uma ferramenta que permite monitorar o ciclo de vida de um agente, suas trocas de mensagens e seus comportamentos que estão sendo executados. Além disso, essa ferramenta permite o controle da execução do agente, no caso, uma execução passo-a-passo ou "slowly" (lentamente). Ele pode ser iniciado por duas maneiras:

- Através da RMA Gui, no menu Tools
- Através da linha de comando:

java jade.Boot intro:jade.tools.introspector.Introspector

Na Figura 3.5 temos a interface visual do Introspector Agent. No lado direito superior, assim como no Sniffer Agent, podemos visualizar todos os agentes registrados na plataforma no instante atual. Na parte inferior, aparecem os endereços dos agentes selecionados. No lado direito, temos todas as informações internas do agente que está sendo depurado. Na guia *Incoming Messages* temos as mensagens que estão sendo enviadas a este agente.Na guia *Outgoing Messages*, ficam as mensagens que foram enviadas por este agente. Vale ressaltar que todas essas mensagens podem ser visualizadas clicando com o botão direito em cima delas. Na parte inferior direita são mostrados todos os *behaviours* que fazem parte do agente em questão, com informações detalhadas tais como o nome da classe, o tipo entre outras.

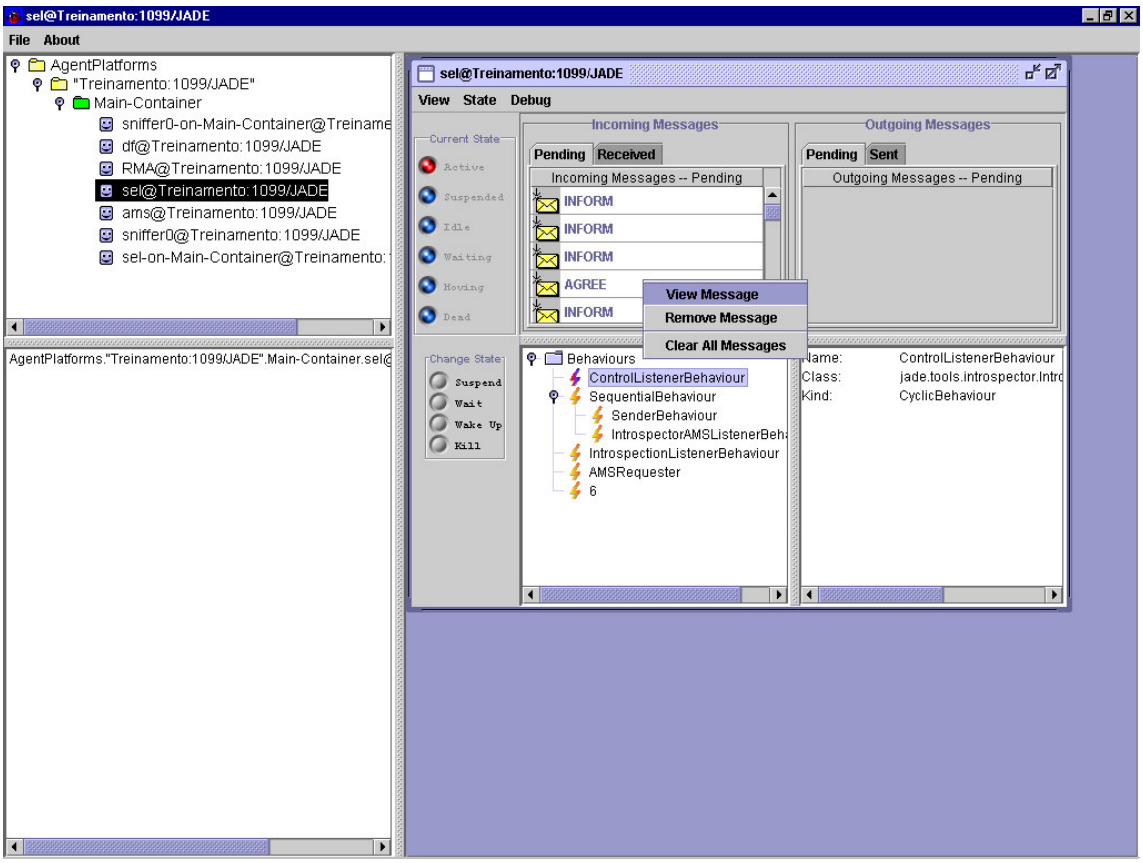

Figura 3.5. Introspector Agent

3.3.5 DF GUI

DF GUI é uma ferramenta para uma interação com DF do JADE, mas que pode ser utilizada com outros DF caso o usuário deseje. Com ela é possível criar uma complexa rede de domínios e sub-domínios de páginas amarelas com uma simples maneira de controlá-las. Ela permite registrar/cancelar registros/modificar/procurar agentes e serviços e até mesmo integrar DF´s. Pode ser carregada no menu *tools* do RMA Gui. Ela é visualizada no host onde a plataforma, leia-se *main-container,* está sendo executada.

Com essa GUI, o usuário pode interagir com o DF de diversas maneiras tais como visualizar e editar as descrições dos agentes registrados e fazer buscas, ações que se tornam muito úteis em sistemas multiagentes com grande quantidade de agentes.

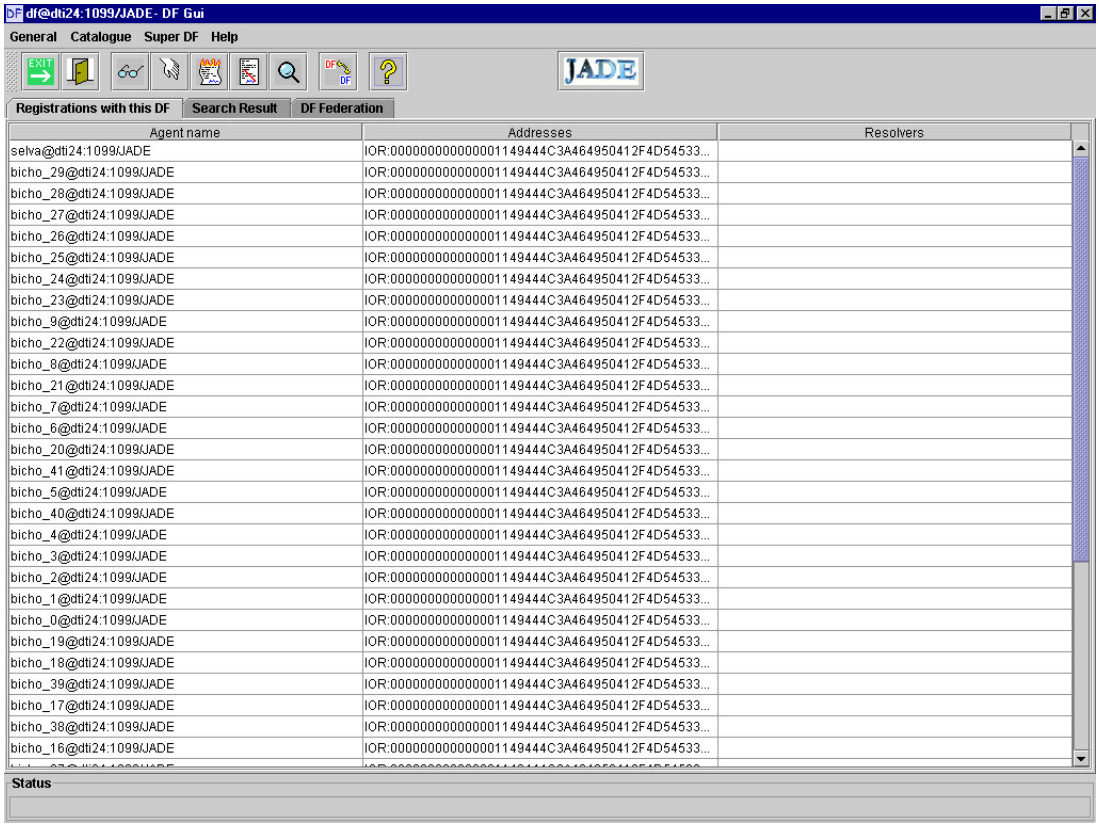

Figura 3.6. DF GUI

Na Figura 3.6, temos a interface visual do DF Gui. Ela é dividida em três painéis. Esses painéis são:

1 **Registrations with this DF** - Mostra uma tabela com todos os agentes registrados nesse DF.
2 **Search Result** - Mostra uma tabela listando descrições de agentes os quais foram especificados na ultima operação de busca nesse DF.
3 **DF Federation** - Mostra a "federação". Ou seja, mostra todo o relacionamento deste Df com outros, onde a tabela *parents* mostra todos os DF´s os quais esse DF está "federado" ou associado. Na tabela *children* mostra todos os DF´s que estão registrados com esse DF.

As funções básicas variam um pouco, uma das outras, de acordo com o painel em que esteja sendo visualizado. Basicamente, elas têm as seguintes finalidades:

1 **View** - Serve para a visualização da descrição do agente selecionado na tabela.
2 **Modify** - Serve para a modificação da descrição do agente selecionado na tabela.
3 **Register new Agent** - Registra um agente no DF.
4 **Deregister** – Cancela o registro o agente selecionado na tabela.
5 **Search** - Serve para fazer uma busca por descrições de agentes neste DF. Faz-se necessário o preenchimento de dois parâmetros pelo usuário: a profundidade da propagação da busca em DF´s associados e o número máximo de descrições de agentes retornadas. Se for deixado em branco será procurado apenas no DF local e todos os agentes serão retornados. Por fim, o usuário deve inserir uma descrição do agente para a busca, caso contrário, a ação de busca retornará todos os agentes ativos que estão no momento registrados no DF.
6 **Federate** - Permite a "federação" ou associação deste DF com outro DF. O usuário deve fornecer o AID do DF o qual deseja associar e sua descrição.

3.4 ESTRUTURA DE UM PROGRAMA EM JADE

Na programação de agentes em JADE, têm-se basicamente três estruturas que são fundamentais:

- Classes de Agentes
- Classes de Mensagens
- Classes de Comportamentos

3.4.1 Agente

Na estrutura de uma classe *Agent*, o método *setup* () é o mais importante. É nele que deve ser implementado o código de configuração inicial do agente, tais como construir e adicionar comportamentos, registrar ontologias, registrar o agente no DF, etc. Segue abaixo, na Figura 3.7, uma estrutura básica de uma classe que deriva da classe *Agent.*

```
public class AgenteJade extends jade.core.Agent {
        public  AgenteJade() {
        }
        /** Método de inicialização do agente. */
        protected void setup() {
                // Aqui fica o código de inicialização do agente:
                // Por exemplo,
             //  adicionar o comportamento do agente,
             // registrar linguagens para ontologias,
                //    registrar o agente no Directory Facilitator...
                //→  addBehaviour(new <Comportamento>());
        }
}
```

Figura 3.7 Estrutura básica de uma classe que implemente um agente JADE

3.4.2 Mensagens

Na estrutura de mensagens, dois importantes aspectos são a classe *ACLMessage* e classe *AID*. Essas classes são fundamentais para a criação de uma mensagem. Na mensagem o remetente é sempre um agente, o receptor é um agente ou um conjunto de agentes, sendo que todos os agentes obrigatoriamente têm que ter um AID. Além disso, deve-se informar o conteúdo da mensagem e a performativa (INFORM, AGREE, QUERY, etc). Na Figura 3.8 temos a estrutura exemplificada na Figura 3.7 com a funcionalidade de enviar mensagem.

```
import jade.core.Agent; // Biblioteca da classe Agent
import jade.lang.acl.ACLMessage; // Biblioteca da classe ACLMessage
import jade.core.AID; // Biblioteca da classe AID

public class AgenteJade extends Agent {

        public  AgenteJade() {
        }
        /** Método de inicialização do agente. */
        protected void setup() {
                // Aqui fica o código de inicialização do agente:
                // Por exemplo,
             // registrar adicionar o comportamento do agente,
             // registrar linguagens para ontologias,
                //    registrar o agente no Directory Facilitator...
                //→  addBehaviour(new <Comportamento>());


                ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                msg.setSender(getAID()); // getAID <- Retorna o AID do Agente
                msg.addReceiver(new AID("AgenteReceptor", AID.ISLOCALNAME));
                msg.setContent("Minha Primeira Mensagem");
                send(msg);
        }
}
```

Figura 3.8 Estrutura básica de uma classe que envia uma mensagem

3.4.3 Comportamentos

Definir comportamentos de um agente é praticamente definir que ações ele deve tomar em determinadas situações. Ou seja, é uma das funções mais importantes no desenvolvimento de sistemas multiagentes, pois é nelas que vão ser definidos os comportamentos que esses agentes terão no ambiente. Como cada comportamento em JADE é

uma classe (ou *Behaviour* ou derivada de *Behaviour)*, definir estes comportamentos em JADE significa implementar essas classes Java. Na Figura 3.9 temos uma estrutura básica de um comportamento do tipo *OneShotBehaviour.*

```
// OneShotBehaviour = comportamento executado uma única vez
public class Comportamento1 extends OneShotBehaviour {
        /** Construtor. */
        public Comportamento1() { }

        /** Método que implementa o que faz o comportamento. */
     public void action() {
        System.out.println("Essa mensagem será mostrada uma única vez");
        }
}
```

Figura 3.9 Estrutura básica de um comportamento em JADE

Na Figura 3.10 mostra como podemos adicionar à classe Agent o comportamento descrito na Figura 3.9.

```
import jade.core.behaviours.*; // Biblioteca das classes de Behaviours
import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.core.AID;
public class AgenteJade extends Agent {
        public  AgenteJade() {
        }
        protected void setup() {
                                addBehaviour(new Comportamento1());
                }
}
```

Figura 3.10 Adicionando um Behaviour a um agente.

Para criar comportamentos mais complexos, podemos usar por exemplo:

- SequentialBehaviour - Como já foi falado, nele comportamentos são executados em seqüência.

```
SequentialBehaviour comp = new SequentialBehaviour();
comp.addSubBehaviour(new Comportamento1());
comp.addSubBehaviour(new Comportamento2());
//...
comp.addSubBehaviour(new ComportamentoN());
```

Figura 3.11 SequentialBehaviour

No caso da Figura 3.11, do comportanto1 até o comportamentoN serão executados seqüencialmente, aonde o comportamento seguinte só começa quando o anterior estiver terminado. Isso significa que, na Figura 3.11, o comportamento *comp* só termina quando o último sub-comportamento terminar.

- ParallelBehaviour **-** Conforme também já foi mencionado, nesse comportamento, sub-comportamentos podem ser executados em "paralelo".

```
ParallelBehaviour comp = new
ParallelBehaviour(ParallelBehaviour.WHEN_ALL);
comp.addSubBehaviour(new Comportamento1());
comp.addSubBehaviour(new Comportamento2());
//...
comp.addSubBehaviour(new ComportamentoN());
```

Figura 3.12 ParallelBehaviour

No caso da Figura 3.12, os comportamentos são executados em qualquer tempo, e a constante *WHEN_ALL* indica que o comportamento *comp* só termina quando todos os sub-comportamentos tiverem terminado. Vale lembrar da constante *WHEN_ANY*, que força o comportamento encerrar quando qualquer um dos sub-comportamentos terminar e do caso de passar um valor no construtor indicando a quantidade de sub-comportamentos que devem ter encerrado para que o comportamento principal termine.

4. ESTUDO DE CASO

4.1 O CASO: PRESA X PREDADOR

O problema da presa e do predador originou-se baseado em um conto chamado "Red October", no qual um submarino soviético tenta escapar de torpedeiros americanos [FERBER 1999]. Ele consiste de um ambiente composto por dois tipos de animais (presas e predadores), dos quais o predador se alimenta da presa e por isso caça-as. Trazendo essa situação para o ambiente de programação, temos agentes presas (submarinos ou herbívoros) e agentes predadores (torpedeiros ou carnívoros), movendo-se em um espaço comum no formato de um *grid,* aonde o objetivo e fazer com que os agentes Predadores capturem os agentes Presas cercando-as (Veja Figura 4.1).

Figura 4.1. Presa cercada por 4 predadores

Segue abaixo algumas características do problema [FERBER 1999]:

- As dimensões do ambiente são limitadas (finitas).
- Os predadores e as presas se movem em velocidade fixas.
- As presas se movem de maneira randômica, selecionando a próxima direção randomicamente a cada passo.

4.2 VISÃO GERAL DA APLICAÇÃO

O objetivo maior do programa foi demonstrar as funcionalidades básicas do JADE no desenvolvimento, administração e depuração de sistemas multiagentes. Sendo que o problema da presa e do predador adequou-se a esse objetivo devido a sua simplicidade.

No caso, dois tipos de agentes existem inicialmente: agente Bicho e agente Selva. O agente Selva é um agente que serve como repositório e controle das estruturas da selva, dos bichos que nela habitam e dos movimentos nela possíveis. Por ser o ambiente, deve ser único e comum a todos os agentes. O agente Bicho e um agente simples que basicamente tem um único comportamento: mover-se aleatoriamente pelo ambiente. Já o agente Predador e uma classe derivada da classe Bicho, que implementa algumas funcionalidades a mais como um movimento não aleatório que tenta perseguir presas.

Toda a comunicação dos agentes e feita por troca de mensagens ACL implementadas por classes que o JADE oferece. Sendo definido tipos de mensagens para a correta interpretação de seus significados.Alem disso, são implementados comportamentos (behaviours) que são responsáveis pelos principais ações dos agentes como movimentação e recepção de mensagens.

Basicamente, o funcionamento do programa ocorre da seguinte forma: o agente Selva cria agentes Bichos e Predadores de acordo com a quantidade especificada pelo usuário e os dispõe aleatoriamente no ambiente. Estes passam a executar seus respectivos comportamentos.

O processo de movimentação dos agentes Bicho e Predador no ambiente e feito totalmente por trocas de mensagens entre estes e o agente Selva. No caso do agente Bicho ele e feito da seguinte forma:

- escolhe uma direção aleatória a seguir.

- Pergunta ao agente Selva se essa posição e valida através de mensagens
- Caso seja valida, o agente Selva envia mensagens confirmando, efetua o movimento e o agente Bicho atualiza sua posição interna
- Caso seja invalida, o agente Selva envia mensagem cancelando movimento e o agente Bicho cancela a intenção de movimento.
- Repete-se o ciclo.

No caso do agente Predador, já há diferenças uma vez que este não se move sempre aleatoriamente :

- Primeiramente o agente Predador envia uma mensagem passando sua posição para o agente Selva.
- O agente Selva responde com uma mensagem contendo todos os objetos que estão no alcance visual (2 quadros) do agente Predador.
- Caso haja algum agente Bicho no alcance visual do agente Predador, este define seu próximo movimento em direção ao bicho mais próximo.
- Caso não haja agentes Bicho no alcance visual do agente Predador, este escolhe aleatoriamente seu próximo movimento.
- Pergunta ao agente Selva se essa posição é valida através de mensagens
- Caso seja valida, o agente Selva envia mensagens confirmando, efetua o movimento e o agente Predador atualiza sua posição interna
- Caso seja invalida, o agente Selva envia mensagem cancelando movimento e o agente Predador cancela a intenção de movimento.
- Repete-se o ciclo

4.3 ESTRUTURA DO PROGRAMA

O programa é composto por 7 arquivos sendo eles:

- Bicho.java
- Predador.java
- Selva.java
- TelaControle.java
- SelvaGui.java

- CarregaAgente.Java
- AJade.Java

4.3.1 Bicho.java

No arquivo Bicho.Java temos a implementação da classe Bicho.Ela é uma classe derivada da classe Agent do JADE e implementa as funcionalidades da presa no problema da presa x predador. Além de implementar a presa, a classe Bicho serve de base para classe Predador uma vez que ela é uma classe derivada da classe Bicho.

Na classe Bicho existem dois comportamentos (behaviours). Um cíclico (*CyclicBehaviour*) que é responsável pela verificação das mensagens recebidas. E um comportamento *OneShotBehaviour* que executa os movimentos a cada mensagem de posição permitida que o agente recebe.

Em suma, um agente Bicho simplesmente escolhe uma posição aleatória através de uma função randômica que retorna quatro direções possíveis, envia uma mensagem para o agente ambiente Selva, passando sua posição atual e a posição que pretende se mover e aguarda a resposta da mensagem autorizando ou não o movimento.

Na figura 4.2 temos a representação visual da presa implementada pela classe Bicho na aplicação visual.

Figura 4.2 Presa

Foi usado do ambiente JADE a classe ACLMessage para envio de mensagens para o agente ambiente Selva, a classe CyclicBehaviour que é o comportamento cíclico de recebimento de mensagens e a classe OneShotBehaviour que é o comportamento de definir próximos movimentos. Além disso, o agente foi registrado no DF utilizando as classes providas pelo JADE: ServiceDescription, DFAgentDescription e DFService.

4.3.2 Predador.java

No arquivo Predador.java temos a implementação da classe Predador. A classe Predador é uma classe derivada da classe Bicho, trazendo funcionalidades a mais. Além da movimentação aleatória que a classe Bicho implementa, na classe Predador uma movimentação intencional. Ou seja, não randômica. No caso, o agente predador instanciado dessa classe Predador irá movimentar-se em direção à alguma presa que esteja no seu campo de visão. No programa foi definido que o campo de visão do predador seria circular e de alcance de 2 quadros no *grid*. Veja nas Figuras 4.3, 4.4 e 4.5 a representação visual do agente, o campo de visão e os quadros "visíveis" delimitados. Na Figura 4.4, somente a presa dentro da área delimitada pela linha preta está sendo "visualizada" pelo agente predador. As outras presas e estruturas, por não estarem no campo de visão não estão sendo "enxergadas" pelo agente. Na figura 4.5, comprovamos que a visão delimitada pelos quadros na figura 4.4 é circular.

Figura 4.3 Representação Visual do agente Predador

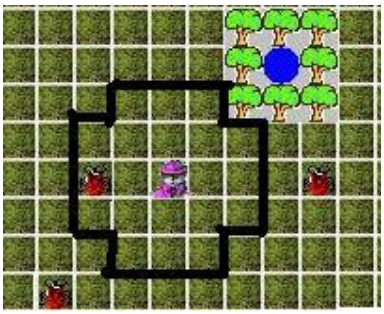

Figura 4.4 Delimitação dos campos visíveis do agente Predador

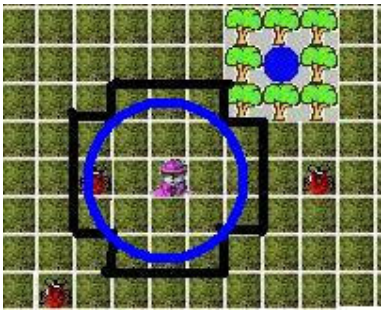

Figura 4.5 Visão Circular do agente Predador

Caso não haja nenhum agente Bicho no campo de visão do agente Predador, este irá mover-se com um agente Bicho, ou seja, aleatoriamente.

A diferença na implementação está no fato que antes de mover-se aleatoriamente, o agente Predador envia uma mensagem ACL para o agente ambiente Selva passando sua posição. O ambiente responde a mensagem contendo *strings* que representam os objetos que estão no campo de visão do agente Predador. Caso haja algum agente Bicho no campo de visão, o agente Predador tentará movimentar-se em direção ao agente Bicho tentando aproximar-se.

Em relação a JADE, praticamente as mesmas classes com as mesmas implementações da classe Bicho foram utilizadas na classe Predador.

4.3.3 Selva.Java

No arquivo Selva.java temos a implementação da classe Selva, que é a classe que implementa o agente ambiente Selva. Ela é derivada da classe *Agent* do JADE e é responsável pela instanciação dos agentes Predador e Presa, controle e validação da movimentação destes agentes, chamada e atualização da interface visual.

Por ser o agente ambiente, o agente Selva deve ser comum à todos os agentes Presa e Predador, a fim de que evite-se ambigüidades uma vez que o agente Selva mantêm um matriz de caracteres que representam os objetos (agentes ou estruturas) e suas respectivas posições no momento.

No ambiente Selva, apenas 3 objetos diferentes são possíveis são agentes Presa, agentes Predador e estruturas físicas. Na matriz os agentes Presa estão representados pela letra ‘B’ (de bicho), os agentes Predador pela letra ‘P’, e as estruturas físicas geralmente por letra ‘E’ (Há algumas variações com intuito somente visual, como por exemplo, letra ‘A’ para árvores – Ver Figura 4.6).

Agentes só podem “movimentar-se” em espaços livres. Espaços ocupados por outros agentes ou estruturas não são possíveis de ser ocupados por outros agentes simultaneamente. Logo, espaços livres foram representados na matriz pela letra ‘R’ (de rua).

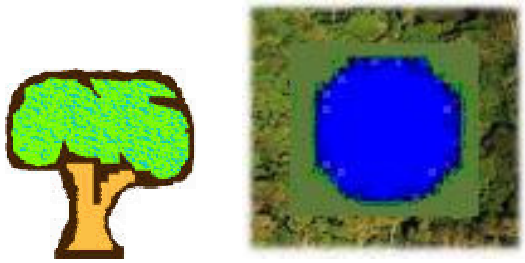

Figura 4.6 Estruturas da Selva

Além do controle da matriz, o agente Selva responde a determinadas mensagens que os agentes enviam para ele. Entre elas, estão as mensagens de introdução à Selva, (“OLA” e “OLA_SOU_PREDADOR”), das quais o agente Selva que identifica se determinado agente é Presa ou Predador. Além disso, envia mensagens confirmando a movimentação dos agentes ou cancelando (“POSICAO_INVALIDA”). Segue abaixo a descrição dos principais métodos do agente Selva:

- **private void carregaTela ()** – método que instancia as interfaces visuais de controle e de apresentação.
- **protected void convidarBichos(int numBichos) –** método que instancia quantidade “n” de agentes Bicho.
- **protected void convidarPredadores(int numPredadores) –** método que instancia quantidade “n” de agentes Predador.
- **protected void finalizaSelva () –** método que finaliza o próprio agente Selva.
- **protected void fecharSelva () –** método que envia mensagens para todos os agentes registrados informando que informando que o ambiente será finalizado.

- **private int[] posicionaBicho (char tipoBicho) -** esse método é usado na primeira vez que um agente se comunica com o ambiente onde é retornado uma posição aleatória e válida para o agente Presa ou Predador.
- **private void informaPosicao (AID idBicho, char tipoBicho)-** método que envia uma mensagem ACL contendo a posição retorna pelo método posicionaBicho ().
- **private void validaPosicao (ACLMessage msgPosicao, char tipoBicho) -** método que valida uma posição e envia uma mensagem ACL informando se esta posição é válida ou não.
- **protected void respondeVisao (ACLMessage msgVisao) –** método que envia uma mensagem ACL para um agente Predador contendo os objetos que estão dentro do campo de visão deste agente.
- **private String visaoBichoNaPosicao (int i, int j) –** essa função retorna uma string de caracteres que representam os objetos que circundam na posição informada. Ou seja, retorna os objetos visualizáveis do agente Predador numa determinada posição.

Em relação à JADE, o agente Selva utiliza um *behaviour* do tipo *CyclicBehaviour*, que é responsável pelo recebimento de mensagens. Há também, uso da classe *AID*, não só para criação do seu próprio AID mas como também dos AIDs dos outros agentes uma vez que o agente Selva que é o responsável pela instanciação dos outros agentes. Sem falar nas muitas utilizações da classe *ACLMessage* pois o envio de mensagens por parte deste agente é muito constante.

Na figura 4.7 temos a representação visual da Selva, com as estruturas tais como árvores e lagoas. Lembrando que somente as áreas permitidas para movimentação são as áreas verdes na figura (‘R’ na matriz).

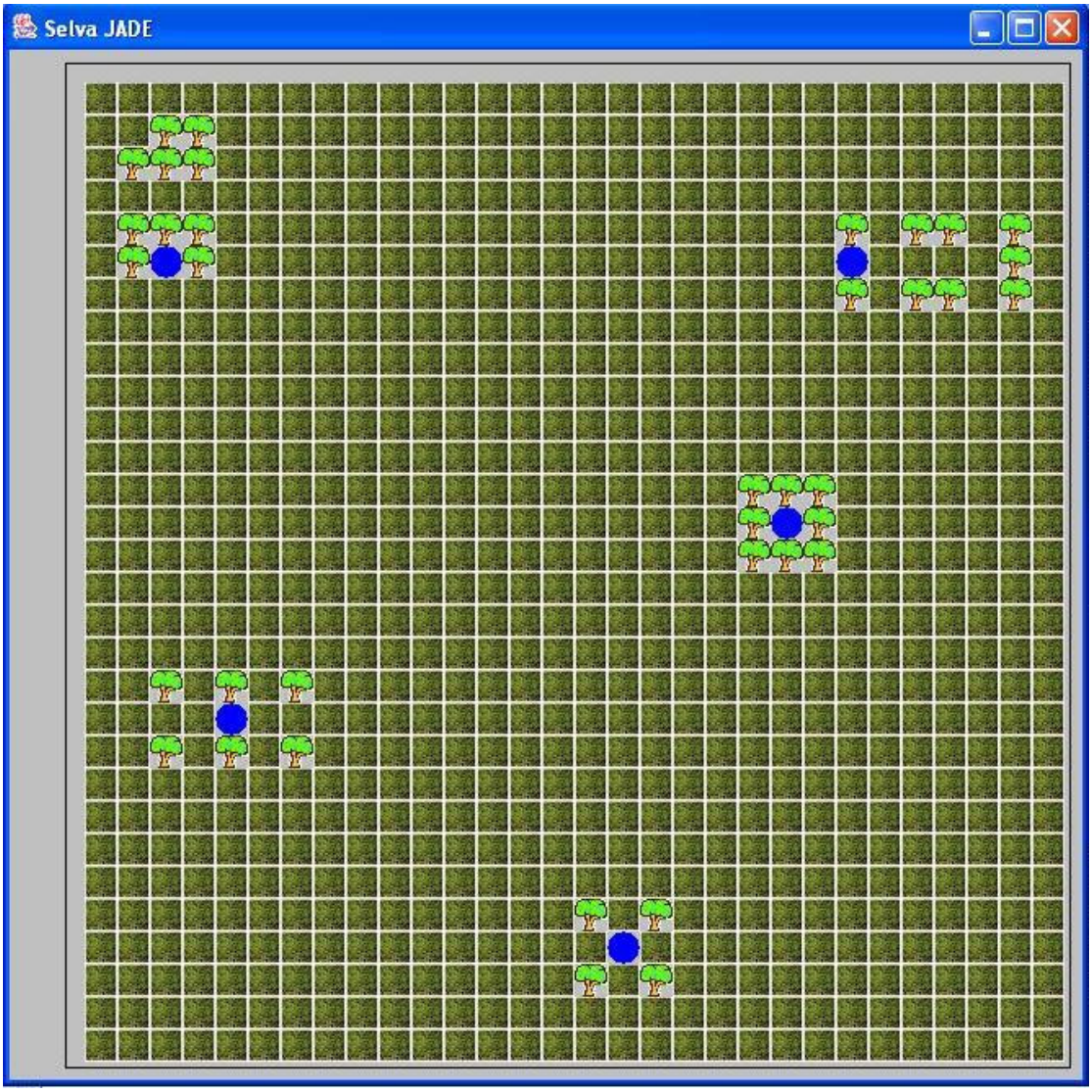

Figura 4.7 Representação Visual da Selva

4.3.4 TelaControle.java

No arquivo TelaControle.java temos apenas a implementação de uma tela de controle em que o usuário pode especificar a quantidade de agentes Bicho e agentes Predador que irão popular a selva e botões de controle de início e término da aplicação – Ver Figura 4.8.

Figura 4.8 Interface de Controle

4.3.5 SelvaGui.java

A classe SelvaGui desenha em uma janela toda a interface visual do ambiente Selva (Veja Figura 4.7). Inicialmente, ela lê a matriz do agente Selva e a "plota" na janela. Após isso ela atualiza essa plotagem sempre que é requisita pelo agente ambiente através do método atualiza ().

4.3.6 CarregaAgente.java

A classe CarregaAgente implementa métodos que instanciam qualquer agente. Essa classe utiliza bibliotecas fornecidas pelo JADE que automatizam essa criação. O método startJade (), instancia e carrega o ambiente JADE e o RMA Gui. Já o método startAgente cria agentes com argumentos ou não, informando seu nome e o caminho da classe que implementa este agente determinado.

4.3.7 AJade.java

Essa classe serve apenas para iniciar toda aplicação. No seu método principal, a classe CarregaAgente é chamada para instanciar o agente Selva e a plataforma JADE. Na Figura 4.9 temos a visualização da aplicação em execução.

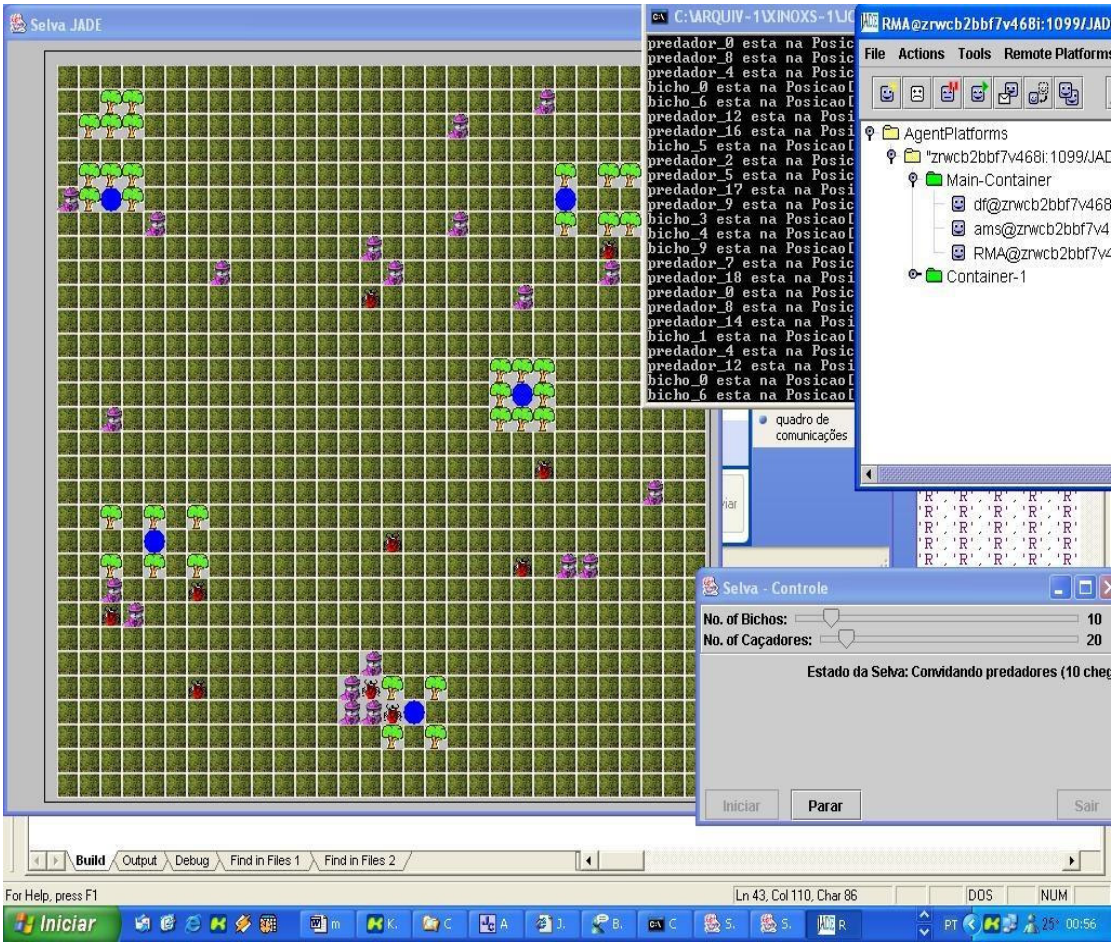

Figura 4.9 Aplicação em execução

CONCLUSÃO

Podemos afirmar que sistemas multiagentes têm evoluído e mostrado sinais claros de que possuem um enorme potencial computacional. A aplicabilidade da tecnologia de agentes vem crescendo rapidamente o que deixa margem para surgimento de ambientes multiagentes variados, necessitando de uma padronização para manter um grau de interoperabilidade.

Conclui-se também da necessidade de frameworks que auxiliem o desenvolvimento de agentes, tendo JADE como um excelente referencial. Isso porque JADE oferece um ambiente, robusto, seguro, conciso e abstrai, ao desenvolvedor de agentes, da necessidade de preocupar-se com a implementação de uma plataforma eficiente de agentes, bem como a comunicação, troca de mensagens e muitos outros atributos que um sistema multiagente necessita. Além disso, oferece uma gama de ferramentas de monitoração, gerenciamento e depuração que ajudam tanto no desenvolvimento quanto na manutenção e suporte de sistemas multiagentes. Além disso, vale ressaltar a grande preocupação que JADE tem em sempre manter os padrões especificados pela FIPA. Fato que aumenta o grau de interoperabilidade e escalabilidade do ambiente em relação às outros sistemas multiagentes.

Pelo fato de a tecnologia de desenvolvimento de agentes ser relativamente nova, muitas questões ainda precisam ser resolvidas e há muito ainda a ser explorado. Porém, por ser uma ferramenta gratuita e de código aberto e por está em constante atualização, JADE se torna uma excelente alternativa para o desenvolvimento e suporte de sistemas multiagentes inteligentes.

BIBLIOGRAFIA

BAKER, Albert. **JAFMAS – A java-based agent framework for multiagent systems. Development and Implementation.** Cincinnati: Department of Electrical & Computer Engineering and Computer Science University of Cincinnati, 1997. Doctoral thesis.

BASTOS, Núbia M.G. **Metodologia Científica**. [S.n.t]

BELLIFEMINE Fabio, CAIRE Giovanni, TRUCCO Tiziana, RIMASSA Giovanni. **JADE Programmer's Guide**. 2003 Disponível em http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf

BIGUS, Joseph P. BIGUS Jennifer. **Constructing Intelligent Agents Using Java** – Second Edition – Wiley Computer Publishing 2001

COULOURIS, George & DOLLIMORE, Jean & KINDBERG, Tim. **Distributed Systems – Concepts and Design**. Third Edition. Addison – Wesley Publishers. 2001.

DEITEL, H. M. & P. J. Deitel. **Java: Como Programar** 3a. Edição 2000

FERBER, Jacques. **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**, Addison-Wesley Pub Co; 1st Edition, February 25 1999.

FILHO, José Eurico de Vasconcelos. **Multiagentes Inteligentes em simulação – Uma Aplicação na segurança pública.** Fortaleza,UNIFOR,2002. Monografia de conclusão de curso. Informatica Graduação.

FRANKLIN, S.; A. **Is It na Agent or Just a Program? A Taxonomy for Autonomos Agents**. Nova York: Springer-Verlag 1996.

GUDWIN, Ricardo R. **Introdução à Teoria de Agentes.** DCA-FEEC-UNICAMP. http://www.dca.fee.unicamp.br~/~gudwin/courses/IA009/

HEILMANN, Kathryn, Dan & Light, Alastair & Musembwa, Paul.. **Intelligent Agents: A Technology and Business Application Analysis**. http://www.mines.u-nancy.fr/~gueniffe/CoursEMN/I31/heilmann/heilmann.html

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java 2 – Volume 1 - Fundamentos**. 7ditora Makron, 1 Edicao, 2000

JADE, Cselt http://sharon.cselt.it/projects/jade Parma - Italia

JENNINGS, Nicholas R., SYCARA, Katia. WOOLDRIDGE, Michael. **A Roadmap of Agent Research and Development.** 1998 Kluwer Academic Publishers

LUFT, Celso Pedro. **Minidicionário Luft**. 14ª. Edição. 1998. Editora Ática. São Paulo.

MENESES, Eudenia Xavier. **Jornada de Atualização em Inteligência Artificial – Integração de Agentes de Informação.** 2001. http://www.ime.usp.br/~eudenia/jaia/

MICROSOFT AGENT, http://www.microsoft.com/msagent/

RUSSEL, Stuart J.; PETER Norvig. **Artificial Intelligence: A Modern Approach**. Englewood Cliffs, Nova Jersey: Prentice Hall, 1995.

SILBERSCHATZ, Abraham & GALVIN, Peter & GAGNE, Greg. **Sistemas Operacionais – Conceitos e Aplicações.** 1ª. Edição. Editora Campus. 2000.

SILVEIRA, Ricardo A. **Modelagem Orientada a Agentes Aplicada a Ambientes Inteligentes Distribuídos de Ensino: JADE –** ***Java Agent Framework for Distance learning Environments***. Tese (Doutorado) – Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS - 2000. Disponível em: http://www.inf.ufrgs.br/~rsilv/publicacoes/RicardoTese.pdf

TORSUN, I.S. **Foundations of Intelligent Knowledge-based system**s.London: Academic Press, 1995.

WEISS, Gerhard. **Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence**, MIT Press, 1999.

WOOLDRIDGE, Michael & Jennings, Nick. Intelligent Agents: Theory and Practice. http://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95.thml

ZEUS. **http://www.labs.bt.com/projects/agents.htm**