5 – Suporte de Software

Neste capítulo apresentam-se ferramentas de suporte para o desenvolvimento de sistemas multi-agentes e que foram utilizadas para o desenvolvimento do *framework* de gerenciamento de contratos, cujo modelo conceitual foi apresentado no Capítulo 4.

5.1 Modelos de Comunicação entre agentes

A forma de comunicação entre agentes é fundamental para a integração de entidades independentes. Em sistemas multi-agentes, 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. Há inúmeros modelos de comunicação entre agentes, sendo o modelo por quadro-negro (*blackboard*) por troca de mensagens os mais utilizados.

Segundo a arquitetura do quadro negro, os agentes se comunicam entre si de maneira indireta através de um quadro negro (Martin et al, 1999). O quadro negro é uma estrutura de dados persistente em que existe uma divisão em regiões ou níveis, visando facilitar a busca de informações. Ele é um meio de interação entre os agentes que funciona como uma espécie de repositório de mensagens. Nele, os agentes interagem através da escrita e leitura de mensagens.

Pode-se dizer que o quadro negro é uma memória de compartilhamento global em que existe uma quantidade de informações e conhecimentos usados para leitura e escrita pelos agentes. Em sistemas multi-agentes, os quadros negros são utilizados como um repositório de perguntas e respostas. Os agentes necessitam de alguma informação escrevem seu pedido no quadro à espera que outros agentes respondam à medida que acessem o mesmo (Lemos, 2002).

O outro modelo de comunicação, baseado em troca de mensagens é amplamente utilizado para o desenvolvimento de sistemas multi-agentes abertos. O *framework* discutido no capítulo 4 adota o modelo de troca de mensagens como meio de interação entre agentes.

No paradigma de troca de mensagens, é necessário que cada agente conheça os nomes e endereços dos agentes os quais querem se comunicar, para que as mensagens possam ser trocadas. Para simplificar o mecanismo de troca de mensagens, estes sistemas costumam contar com agentes facilitadores que armazenam os endereços e serviços oferecidos pelos agentes do sistema (Helin et al, 2002).

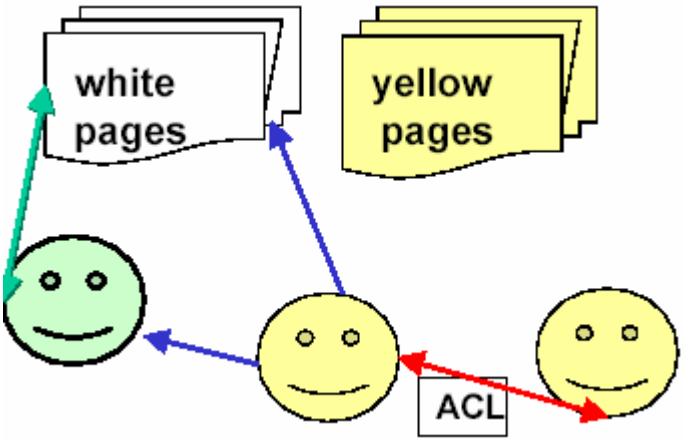

Figura 33: Comunicação entre agentes via troca de mensagens.

Quando um agente necessita se comunicar com outro agente, este pode solicitar ao agente facilitador que encontre a identidade do agente através de seu nome (através das *white pages*) ou pelo tipo do agente ou serviço oferecido (através das *yellow pages*).

A comunicação via troca de mensagens demanda os seguintes itens:

- linguagem para representação de mensagens;
- uma maneira de codificar as mensagens (binário, XML, ASCII);
- uma linguagem de representação de conhecimento (conteúdo da mensagem);
- uma maneira de transportar as mensagens (SOAP, HTTP, IIOP).

O desenvolvimento do *framework* de gerenciamento de contratos utiliza a plataforma JADE, que será discutida na seção 5.3 deste capítulo. JADE emprega os padrões da FIPA (*Foundation for Intelligent Physical Agents)* para o desenvolvimento de aplicações baseada em agentes. A seguir serão discutidas

algumas características da linguagem de comunicação entre agentes FIPA-ACL, utilizada na plataforma JADE.

5.2 Linguagem de comunicação FIPA-ACL

A FIPA_ACL é uma linguagem de comunicação desenvolvida pela FIPA (*Foundation for Intelligent Physical Agents*). A FIPA é uma organização sem fins lucrativos fundada em 1996 na Genebra, Suíça, cujo objetivo é estabelecer padrões para o desenvolvimento de agentes inteligentes. Uma mensagem FIPA_ACL contém um ou vários parâmetros. Os parâmetros da mensagem variam de acordo com o contexto, apenas o parâmetro *performative* é obrigatório em uma mensagem ACL, embora também seja esperado que se utilize os parâmetros *sender*, *receiver* e *content*. A FIPA-ACL organiza os parâmetros de mensagem em 4 categorias: transferência de informação, negociação, ação e gerenciamento de erros (Fipa,2003).

```
(inform
: sender client
: receiver supplier
: content
`payment (product X, 15.00)`
: ontology ecommerce
: in-reply-to 4856A
: conversation-id `4856B` )
```

Figura 34: Exemplo de uma mensagem definida em FIPA-ACL.

A figura 34 ilustra um exemplo da estrutura de uma mensagem FIPA-ACL. Esta mensagem pode representar, por exemplo, o pagamento de um produto de um cliente a um fornecedor. O remetente da mensagem é “client”, o destinatário é “supplier”, o conteúdo é “payment(product X, 15.00)”, a ontologia é “ecommerce”, o campo `in-reply-to` informa que a mensagem está sendo enviada em resposta a mensagem “4856A” e por fim, o campo `conversation-id` informa que o código da mensagem é “ 4856 B”.

No framework de gerenciamento de contratos, é passado no parâmetro `content` da mensagem `FIPA-ACL` o objeto serializado *Message*, que foi descrito na seção 4.3.1 do capítulo 4.

5.3 Plataforma JADE

Os agentes criados para se testar a aplicação foram desenvolvidos utilizando-se da plataforma JADE. JADE é um ambiente para desenvolvimento

de aplicações baseada em agentes conforme as especificações da FIPA (*Foundation for Intelligent Physical Agents*) para interoperabilidade entre sistemas multiagentes totalmente implementado em Java. 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 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.*

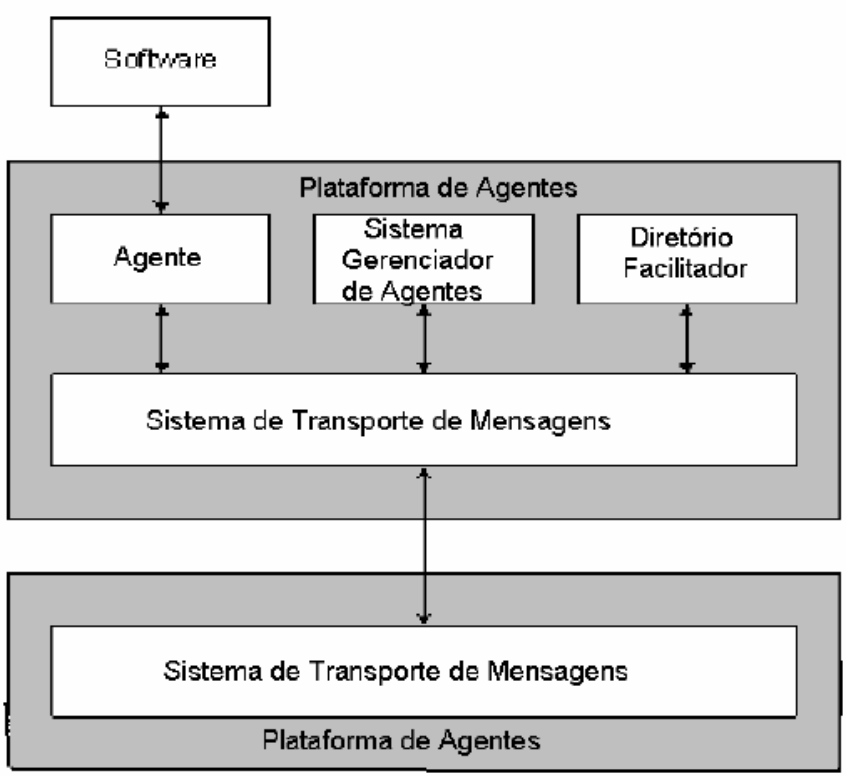

Figura 35: Modelo padrão de plataforma de agentes definido pela FIPA.

O modelo de plataforma padrão especificado pela FIPA, que pode ser visto na figura 35, é composto pelos seguintes estruturas definidas abaixo:

O agente (*Agent*) 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 35, 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 (*whitepages*) e controle de ciclo-de-vida, mantendo um diretório de identificadores de agentes (*Agent Identifier - AID*) e estados de agentes. Ele é o responsável pela autenticação de 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 35, é o agente que provê o serviço de páginas amarelas (*yellow-pages*) dentro da plataforma.

Na parte inferior da figura 35 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.

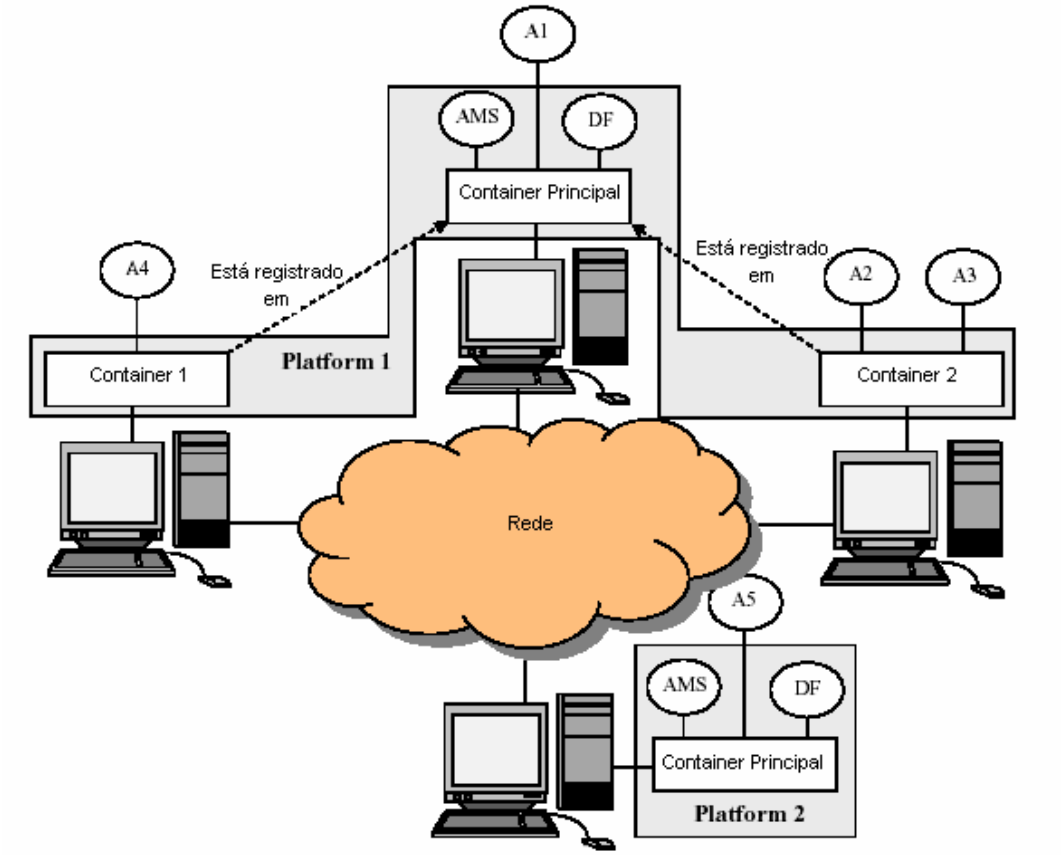

Figura 36: Containers e a Plataforma JADE, extraído de (Jade,2003), adaptado.

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.

5.3.1 O Agente em JADE

FIPA nada especifica sobre as estruturas internas dos agentes, fato que foi uma escolha explícita em conformidade 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).

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).

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.

`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.

JADE possui várias classes de comportamentos prontas para uso pelo desenvolvedor adequando-as de acordo com a necessidade específica do agente. A seguir é mostrado um diagrama de classes em UML, extraído de (Silva,2003) que ilustra 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.

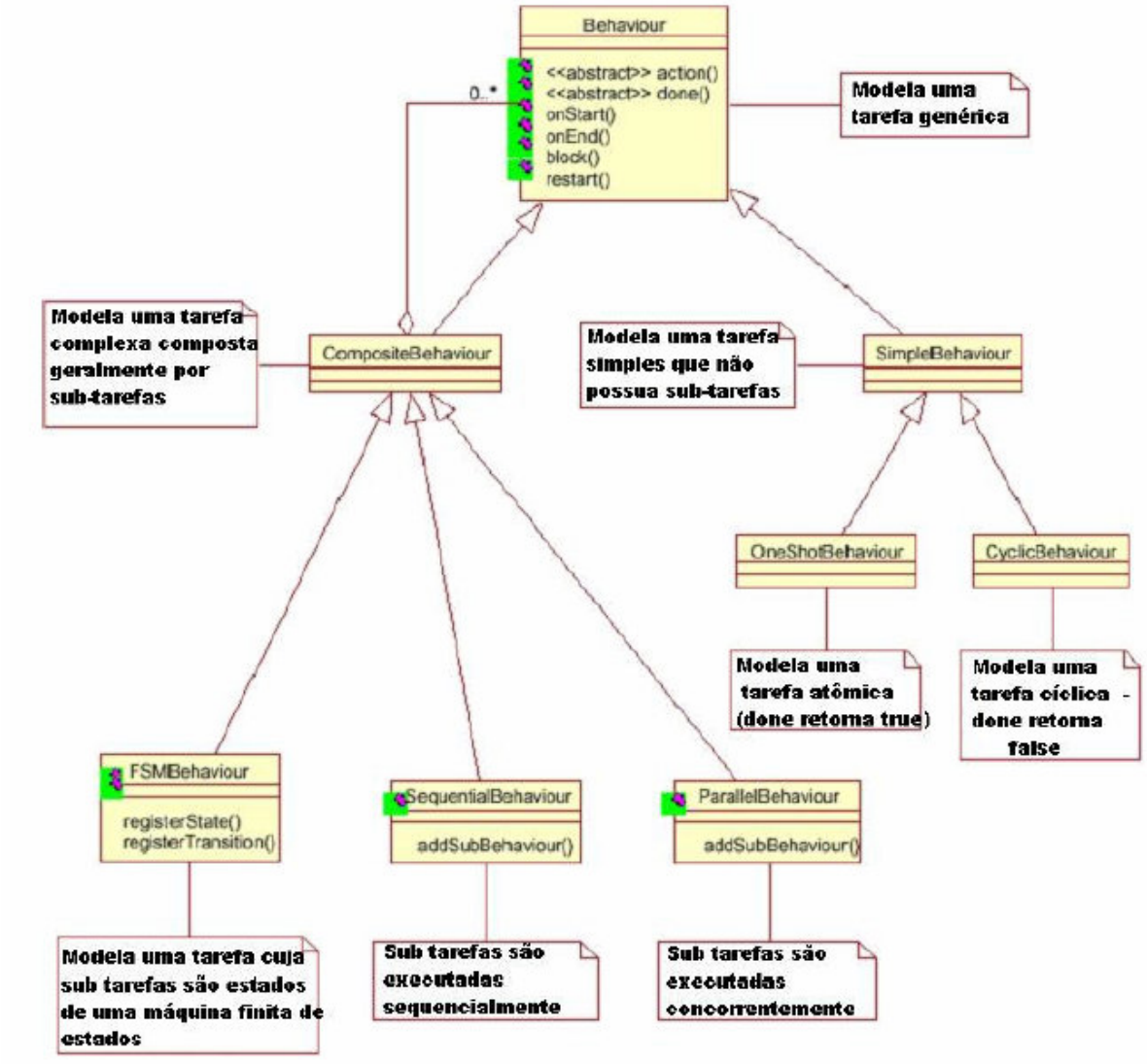

Figura 37: Hierarquia das classes derivadas de Behaviour, extraída de (Silva, 2003).

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

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 (Finite

State Machine). 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:

- Registrar um comportamento único como estado inicial através do método registerFirstState passando como parâmetros o Behaviour e o nome do estado;

- Registrar um ou mais comportamentos como estados finais através do método registerLastState;

- Registrar um ou mais comportamentos como estados intermediários através do método registerState;

- 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 comportamentosfilho 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.

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 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.