Universidade Federal do ABC
UFABC

André Filipe de Moraes Batista
andre.batista@ufabc.edu.br

Manual Complementar do Projeto de Pesquisa:
Sistemas Multiagentes na Construção de um *Middleware* para Suporte a Ambientes Computacionais

Orientação:
Profª Drª Maria das Graças Bruno Marietto

Santo André, Agosto de 2008

Prefácio

> *"Atingimos a perfeição não quando nada pode acrescentar-se a um projeto, mas quando nada pode retirar-se"*
> ——Saint-Exupéry

A idéia de escrever este manual teve início nos meus estudos de iniciação científica na Universidade Federal do ABC, sob a orientação da professora Graça Marietto. É neste espírito de aprendizado que este manual foi escrito. Desde o primeiro momento que descobri que iria trabalhar programando em Java, e que esta programação seria para sistemas multiagentes, a alegria e ansiedade foram estonteantes. É este o meu objetivo: ser o mais didático possível e fazer com que você sinta um pouco das emoções que senti durante as longas horas diante do computador, programando e ao mesmo tempo descobrindo a plataforma JADE e suas façanhas.

A programação de sistemas multiagentes é uma técnica de desenvolvimento de sistemas distribuídos, e neste contexto encontramos questões tais como: "como distribuir?", "como fazer a comunicação entre as partes do sistema?", "como administrar o sistema?", entre outras. Essas e outras perguntas foram respondidas ao longo do estudo desta plataforma. Além de abstrair muitos recursos do programador, as facilidades de implementação fazem com que esta seja uma das mais utilizadas plataformas de desenvolvimento de aplicações distribuídas.

Este manual não está finalizado e nunca estará. Ele contará sempre com um toque especial seu. Sim, você!... Você deverá ao longo de sua leitura criar cenários para suas aplicações, buscar soluções para as dúvidas que surgirão, enfim, nunca se contentar apenas com o que está escrito, tente sempre tirar suas próprias conclusões. Fazendo isto você aprenderá cada vez mais.

André Filipe de Moraes Batista

Aos meus pais.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Instalação

Neste capítulo será demonstrado o processo de instalação da plataforma JADE.

1.1 Requisitos Básicos

Por ser desenvolvida em Java, o requisito mínimo para executar a plataforma JADE é o *run-time* do Java. No caso *Java Run-Time Environment* (JRE) versão 1.4 ou superior.

1.2 *Software* de Instalação da Plataforma JADE

Todos os *software* JADE, inclusive *add-ons* e novas versões, estão distribuídos sob as limitações da LGPL[1] e disponíveis para *download* no *web site* `http://jade.cselt.it` ou `http://jade.tilab.com`.

A versão atual da plataforma JADE utilizada neste trabalho é a 3.5. Para esta versão, cinco arquivos são disponibilizados:

1. **jadeSrc.zip** (1.8 MB) – código fonte do *software* JADE;
2. **jadeExamples.zip** (270 KB) – código fonte dos exemplos de programas feitos em JADE;
3. **jadeDoc.zip** (4.7 MB) – toda a documentação, incluindo o *javadoc* da API da plataforma e manuais de referência;
4. **jadeBin.zip** (2.0 MB) – o *software* JADE binário, pronto para uso;

[1] A licença 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 etc. Já alguns dos deveres são: não fazer modificações privadas e secretas, não alterar a licença do *software* e suas modificações, dentre outros.

5. **jadeAll.zip** (5.8 MB) – todos os arquivos anteriormente citados.

Recomenda-se o *download* do arquivo `jadeAll.zip`. Após o *download* descompacta-se o arquivo interno `jade-Bin-3.5.zip` em um diretório, por exemplo o diretório raiz (`C:`). Uma pasta denominada `jade` é criada. Deve-se incluir as seguintes linhas na variável de ambiente CLASSPATH:

```
.;c:\jade\lib\jade.jar; c:\jade\lib\jadeTools.jar; c:\jade\lib\iiop.jar
```

Com isto a plataforma JADE está instalada e configurada para rodar no ambiente *Windows*.

1.3 Execução do Ambiente

Após os passos citados anteriormente, é possível a execução da plataforma JADE pelo *prompt* de comando. A linha de execução da plataforma JADE é a seguinte:

```
java jade.Boot [opções] [agentes]
```

Esta linha de execução será abordada nas próximas seções com mais detalhes. Neste momento, para verificar se a instalação e a configuração da plataforma foram feitas corretamente, digite no *prompt* o seguinte comando:

```
java jade.Boot
```

Com este comando a plataforma JADE é executada. Se tudo foi configurado corretamente, a seguinte mensagem será exibida:

```
INFO: ----------------------------------
This is JADE 3.5 - revision 5988 of 2007/06/21 11:02:30
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
----------------------------------------
11/11/2007 12:22:50 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
11/11/2007 12:22:50 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
11/11/2007 12:22:50 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
11/11/2007 12:22:50 jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
```

```
11/11/2007 12:22:50 jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
11/11/2007 12:22:50 jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal
.parsers.SAXParser
11/11/2007 12:22:50 jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://lap:7778/acc
11/11/2007 12:22:50 jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@lap is ready.
------------------------------------------
```

A exibição da mensagem `Agent container Main-Container@lap is ready` indica que a plataforma está em execução. Para finalizar a execução pressione as teclas CTRL+C.

Existe um *bug* na versão 3.5 da JADE, onde um erro ligado aos *sockets* é exibido pois a máquina que está executando a plataforma pode não estar em rede. Esta falha só apareceu na versão 3.5 e sua correção ainda está em desenvolvimento. Recomenda-se que quando uma máquina estiver fora da rede deve-se incluir o parâmetro `-detect-main false` na linha de execução da plataforma, da seguinte maneira:

```
java jade.Boot -detect-main false
```

1.4 Integração com a IDE NetBeans

Vamos integrar as bibliotecas da plataforma JADE com a IDE Netbeans[2], para que seja possível o desenvolvimento de agentes contando com as ferramentas de uma IDE. No NetBeans abra o menu Ferramentas (*Tools*) e clique na opção Gerenciador de Biblioteca (*Library Manager*). Uma janela será aberta.

Na janela Gerenciador de Bibliotecas clique no botão Nova Biblioteca (*new library*). Uma janela de diálogo será aberta. No campo `Nome da Biblioteca` digite JADE e deixe marcado o campo `Tipo de Biblioteca` como Bibliotecas da Classe. E clique em OK. A janela de Gerenciador de Bibliotecas agora apresentará a nova biblioteca adicionada.

Nesta mesma janela, na guia *Classpath* clique no botão `Adicionar JAR/Pasta` e adicione os arquivos (localizados em `C:/jade/lib/`): `http.jar, iiop.jar, jade.jar` e `jadeTools.jar`. Na guia `Javadoc` adicione a pasta `C:/jade/doc`. Clique em OK e as bibliotecas JADE estarão integradas ao Netbeans.

Quando estiver desenvolvendo um projeto no NetBeans, vá na guia `Projetos` e selecione o item `Bibliotecas`. Com o botão direito do mouse selecione a opção `Adicionar`

[2]A versão utilizada foi a 6.0, disponível em `http://www.netbeans.org`

`Biblioteca`. Será aberta a janela Adicionar Biblioteca. Nesta, selecione a biblioteca JADE e clique em OK. Com isto, o NetBeans passará a reconhecer os métodos e atributos fornecidos pela plataforma.

A execução de um agente não será pela IDE[3]. Você deverá executar os agentes a partir de um *prompt* de comando.

[3]Existem formas de se executar um agente pela IDE, mas estas não serão abordadas neste manual.

Capítulo 2

Primeiros Programas

Neste capítulo será apresentada a estrutura básica de uma implementação na plataforma JADE, com o uso de exemplos de implementação.

2.1 *Hello World*

Vamos começar com um programa clássico: um agente JADE que imprime na tela a frase *Hello World*. Agentes JADE são definidos como subclasses da classe `Agent` e seu código inicial (o que o agente fará ao ser executado) deve ser colocado dentro do método `setup()`. O código do *Hello World* é mostrado na Caixa de Código 2.1.

Herdando a classe `jade.core.Agent` os agentes já possuem as operações básicas dentro da plataforma, como registro e configuração e outro conjunto de métodos para implementação de comportamentos pré-definidos, como métodos de troca de mensagens.

Código 2.1: HelloAgent.java

```
import jade.core.Agent;

public class HelloAgent extends Agent{

    protected void setup()
    {
        System.out.println("Hello World. ");
        System.out.println("Meu nome é " + getLocalName());
    }
}
```

Devemos estar dentro do diretório dos arquivos `.java` para que este agente seja executado. Uma vez no diretório, deve-se executar as seguintes linhas de comando no *prompt*:

```
javac HelloAgent.java
java jade.Boot Andre:HelloAgent
```

A primeira linha trata da compilação da classe `HelloAgent.java`, gerando o arquivo `HelloAgent.class`. A segunda linha de comando requer um pouco mais de explicação. Basicamente, agentes são como *Java Applets* na medida em que ambos não podem ser executados diretamente: eles são executados dentro de outro programa que provê os recursos necessários para sua execução. No caso dos *applets*, um *browser* ou um *Applet Viewer* é necessário; para agentes JADE o ambiente de execução é fornecido pela classe `jade.Boot`. O parâmetro `Andre:HelloAgent` indica a classe do agente (`HelloAgent`), e provê um nome único ao agente (`Andre`). A classe `Agent` contém o método `getLocalName()`, que retorna o nome do agente em seu *container*.

Com a execução das linhas de comando tem-se o seguinte resultado:

```
INFO: ----------------------------------
This is JADE 3.5 - revision 5988 of 2007/06/21 11:02:30
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
----------------------------------------
.
.
.

INFO: ----------------------------------------
Agent container Main-Container@lap is ready.
--------------------------------------------
Hello World.
Meu nome é Andre
```

É importante lembrar que este agente ainda está em execução. Para finalização do agente e da plataforma pressione as teclas `CTRL+C`.

2.2 Identificadores de Agentes

De acordo com o padrão FIPA cada instância de agente é identificada por um *agent identifier*. Na plataforma JADE um *agent identifier* é representado como uma instância da classe `jade.core.AID`. O método `getAID()` permite obter a identificação completa (global, nome na plataforma, endereço, etc) do agente, armazenando estas informações em uma lista. Um ID global é aquele que representa o identificador do agente em toda a plataforma. Um ID local refere-se ao conjunto de informações que representam o agente em seu *container*.

O nome global do agente é único na plataforma, e possui a seguinte estrutura:

```
<nome_local>@<nome-plataforma>
```

Por exemplo o agente `Pedro`, localizado na plataforma `Escola`, terá seu nome global definido por `Pedro@Escola`.

A classe `AID` disponibiliza métodos para obter o nome local (`getLocalName()`[1]), o nome global (`getName()`) e os endereços de um agente (`getAllAddresses()`). Para visualização destas funcionalidades, considere o código exibido na Caixa de Código 2.2.

Código 2.2: InfoAgentes.java

```
import jade.core.Agent;
import jade.core.AID;
import java.util.Iterator;
public class InfoAgentes extends Agent{

    protected void setup(){
        System.out.println("Hello World. Eu sou um agente!");
        System.out.println("Todas as minhas informações: \n" + getAID());
        System.out.println("Meu nome local é "+ getAID().getLocalName());
        System.out.println("Meu nome global (GUID) é"+ getAID().getName());
        System.out.println("Meus endereços são:");
        Iterator it = getAID().getAllAddresses();
        while(it.hasNext()) {
            System.out.println("- "+it.next());
        }
    }

}
```

Com a execução da seguinte linha de comando:

```
java jade.Boot Ivan:InfoAgentes
```

Temos o seguinte resultado:

```
Hello World. Eu sou um agente!
Todas as minhas informações:
( agent-identifier :name Ivan@lap:1099/JADE
 :addresses (sequence http://lap:7778/acc ))
Meu nome local é Ivan
Meu nome global (GUID) é Ivan@lap:1099/JADE
Meus endereços são:
- http://lap:7778/acc
```

[1]A classe `Agent` também disponibiliza este método.

O nome local do agente é `Ivan`, conforme especificado na linha de comando. Uma vez que não especificamos o nome da plataforma, JADE automaticamente atribui um nome a esta, usando as informações do *host* e da porta do *container* principal (no caso, o *host* é denominado `lap` e a porta do *container* principal é `1099`). Com isso, o nome global (GUID - *Globally Unique Name*) do agente é `Ivan@lap:1099/JADE`.

Embora pareça um endereço, o GUID não é o endereço do agente. No caso, vemos que o endereço do agente Ivan é `http://lap:7778/acc`(Agent Communication Channel). Os endereços incluídos em uma AID são referentes ao MTP (*Message Transport Protocol*). Por padrão, a plataforma JADE atribui um MTP HTTP ao *main-container*. Estes endereços são usados na comunicação entre agentes que estão em plataformas diferentes. Pode-se incluir endereços FTP, IIOP dentre outros.

Para designarmos um nome para a plataforma, devemos passar este nome como parâmetro na execução da seguinte forma:

```
java jade.Boot -name plataforma-de-teste Ivan:InfoAgentes
```

Neste caso demos o nome de `plataforma-de-teste` à nossa plataforma e, com isso, na execução do agente tem-se o seguinte resultado:

```
Hello World. Eu sou um agente!
Todas as minhas informações:
( agent-identifier :name Ivan@plataforma-de-teste:1099/JADE
 :addresses (sequence http://plataforma-de-teste:7778/acc ))
Meu nome local é Ivan
Meu nome global (GUID) é Ivan@plataforma-de-teste
Meus endereços são:
- http://lap:7778/acc
```

2.3 Passando Informações a um Agente

Considere um agente comprador de livros, que deve saber quais livros irá comprar. Vamos criar um agente em que será possível indicar qual livro este agente irá comprar. Seu código está apresentado na Caixa de Código 2.3.

Código 2.3: CompradorDeLivros.java

```
import jade.core.Agent;
import jade.core.AID;

public class CompradorDeLivros extends Agent{

    private String livrosComprar;
```

```
    protected void setup()
    {
        //imprime mensagem de Bem-Vindo
    System.out.println("Olá!!! Eu sou o Agente Comprador "+ getLocalName()
       +" e estou pronto para comprar!");

        //captura o título do livro que comprará, que foi passado como
            argumento de inicialização
        Object[] args = getArguments();
        if(args != null && args.length>0)
        {
           livrosComprar = (String) args[0];
           System.out.println("Pretendo comprar o livro: "+ livrosComprar);
        }else
        {
            //finaliza o agente
           System.out.println("Nao tenho livros para comprar!");
           doDelete(); //invoca a execução do método takeDown()
        }
    }

    protected void takeDown() {
           System.out.println("Agente Comprador" + getAID().getName() + "
              está finalizado");
    }

    }
```

Para executar o Código 2.3 digitamos no *prompt*:

```
javac CompradorDeLivros.java
java jade.Boot Jose:CompradorDeLivros("O-Pequeno-Principe")
```

Com a execução destas linhas tem-se o seguinte resultado:

```
Olá!!! Eu sou o Agente Comprador Jose e estou pronto para comprar!
Pretendo comprar o livro: O-Pequeno-Principe
```

Caso não seja passado nenhum parâmetro na execução do agente, este imprimirá a mensagem:

```
Olá !!! Eu sou o Agente Comprador Jose e estou pronto para comprar!
Nao tenho livros para comprar!
Agente Comprador Jose@lap:1099/JADE está finalizado
```

Observe que a frase `Agente Comprador Jose@lap:1099/JADE está finalizado` utiliza no código o método `getAID().getName()`, que retorna o nome global do agente. Por padrão JADE adiciona o número da porta de comunicação da plataforma (padrão é 1099) e também adiciona a *string* `/JADE` ao final, para indicar que trata-se de um agente JADE. Para finalizar a execução da plataforma pressione as teclas `CTRL+C`.

2.4 Comportamentos

Cada ação que um agente pode realizar é representada como um comportamento deste agente. O código que implementa esta funcionalidade deve estar em uma nova classe, que deve herdar as funcionalidades da classe `jade.core.behaviours.Behaviour`.

Uma vez implementado o código referente ao comportamento, para que este seja executado é necessária a invocação, no corpo de ação do agente, do método `addBehaviour` pertencente à classe `Agent`. Como exemplo considere a classe `MeuAgente`, cujo código está ilustrado na Caixa de Código 2.4.

Código 2.4: MeuAgente.java

```
import jade.core.Agent;
import jade.core.behaviours.Behaviour;
public class MeuAgente extends Agent{

    protected void setup(){
        System.out.println("Olá, eu sou um agente.");
        System.out.println("Estou disparando meu comportamento ...");
        addBehaviour(new MeuComportamento(this));
    }
}
```

Quando um agente da classe `MeuAgente` é inicializado seu método `setup()` é executado. A linha de código `addBehaviour(new MeuComportamento(this))` indica a inicialização de um comportamento que está especificado na classe `MeuComportamento`. O parâmetro `this` indica que o agente que executará este comportamento é o próprio agente que está invocando o comportamento.

Toda classe que especifica o comportamento de um agente deve possuir os seguintes métodos:

- `action()` - neste método incluímos o código referente ao comportamento a ser executado pelo agente;
- `done()` - este método devolve um valor booleano, indicando se o comportamento foi finalizado ou não.

A classe `MeuComportamento` está contida na Caixa de Código 2.5.

Código 2.5: MeuComportamento.java

```
import jade.core.Agent;
import jade.core.behaviours.Behaviour;

public class MeuComportamento extends Behaviour{
int i=0;

public MeuComportamento(Agent a){
    super(a);
}
    public void action() {
          System.out.println("* Olá Mundo! ... Meu nome é " + myAgent.
              getLocalName());
               i=i+1;
    }

    public boolean done() {
  //caso este método retorne TRUE o comportamento será finalizado
        return i>3;
    }
}
```

O comportamento indicado nesta classe fará com que sejam impressas quatro mensagens de texto. Neste código, quando a variável `i` assume um valor maior que 3, o método `done()` retorna `true` e a execução do comportamento é finalizada.

É importante notar a utilização da variável `myAgent`. Trata-se de uma variável nativa de uma classe que herda `jade.core.behaviours.Behaviour`. Como um comportamento tem que ser codificado em outra classe, esta variável oferece a capacidade de acessar todos os métodos e atributos do agente que está executando o comportamento. Isto justifica a utilização do método `super(a)` no método construtor da classe `MeuComportamento`. Através da invocação do `super(a)` é que indicamos o agente que será representado pela variável `myAgent`.

Observe também que a classe `MeuComportamento.java` é filha da classe `Behaviour` (fazemos isto através do comando `extends Behaviour`). A mesma funcionalidade poderia ser alcançada se esta fosse filha da classe `SimpleBehaviour` (e conseqüentemente teríamos `extends SimpleBehaviour`). Logo, `Behaviour` possui a mesma funcionalidade de `SimpleBehaviour`, pois `SimpleBehaviour` é uma classe-filha de `Behaviour`.

Após compilar ambas as classes, podemos executar um agente:

```
java jade.Boot Agent1:MeuAgente
```

O resultado desta execução é o seguinte:

```
Olá, eu sou um agente.
```

```
Estou disparando meu comportamento ...
* Olá Mundo! ... Meu nome é Agent1
* Olá Mundo! ... Meu nome é Agent1
* Olá Mundo! ... Meu nome é Agent1
* Olá Mundo! ... Meu nome é Agent1
```

2.4.1 Execução dos Comportamentos

Um agente pode executar vários comportamentos concorrentemente com o uso de um escalonador. Um comportamento é executado até que seu método `action()` chegue ao fim de sua execução. O escalonador controla a execução dos comportamentos de um agente com as seguintes estruturas de dados:

- Uma fila de comportamentos ativos;
- Uma fila de comportamentos bloqueados.

A Figura 2.1, adaptada de [Bellifemine, Caire e Greenwood 2007], ilustra a interação de um agente com estas filas com o uso de uma análise no ciclo de execução de um agente.

A execução de um agente na plataforma JADE é constituída por três níveis básicos. São eles:

1. Inicialização - Consiste na execução do método `setup()`;
2. Realização da tarefa - Representa o nível de execução dos comportamentos do agente. O escalonador seleciona o primeiro comportamento da fila e executa seu método `action()`. Após a execução deste método, verifica-se a finalização do comportamento no método `done()`. Caso este comportamento ainda não esteja finalizado, o escalonador captura o próximo comportamento da lista de comportamentos ativos, colocando este comportamento ainda não finalizado no final da fila, para ser posteriormente executado, ou seja, este comportamento é bloqueado até que chegue sua vez de ser executado. Quando um comportamento é finalizado, é removido da lista de comportamentos ativos e enviado para a lista de comportamentos bloqueados;
3. Limpeza e finalização - Consiste na execução de métodos específicos para finalização do agente (*e.g* `takeDown()`).

2.4.2 Bloqueando Comportamentos

Com o uso do método `block()` é possível bloquear um comportamento de um agente. Com a execução deste método, o comportamento em execução é movido para a lista

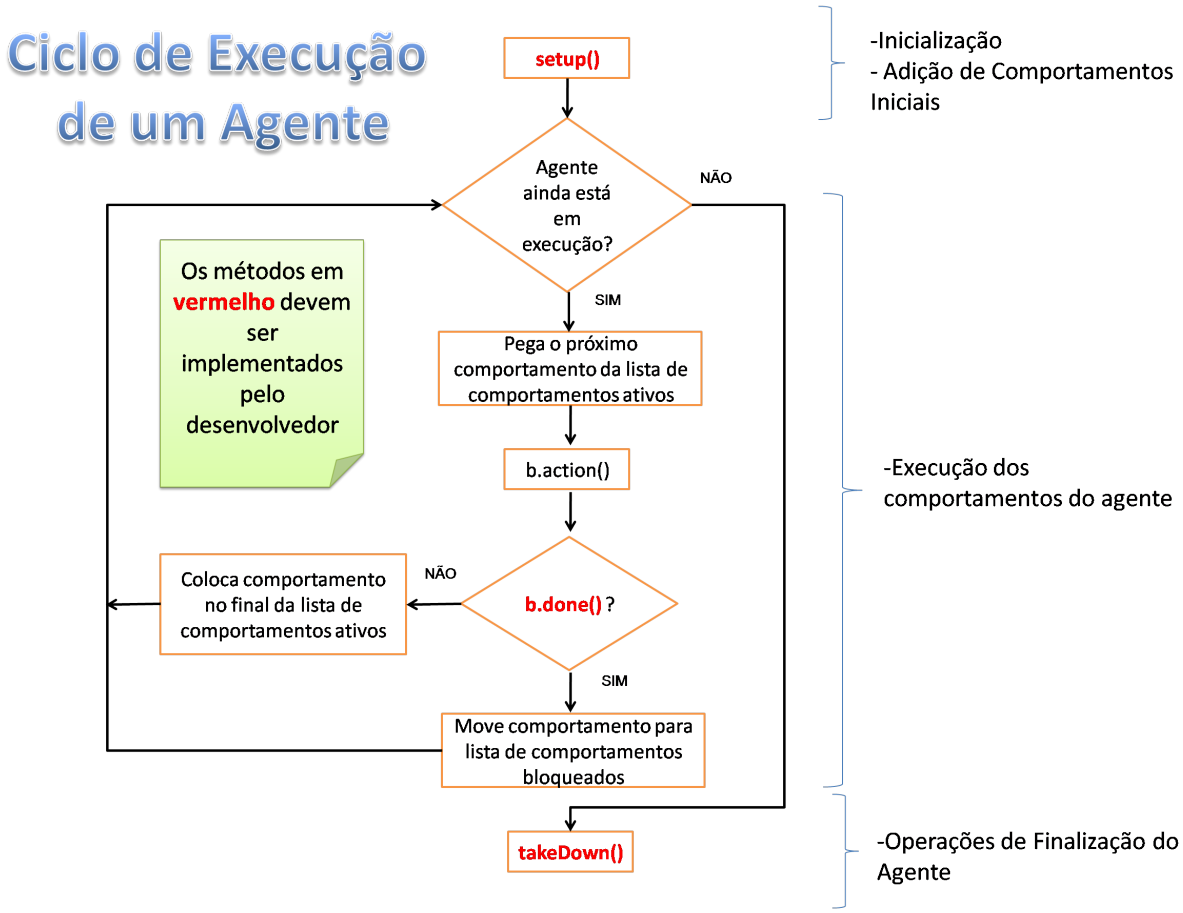

Figura 2.1: Ciclo de Execução de um Agente.

de comportamentos bloqueados até que um evento ocorra. Este método pode ser útil na recepção de mensagens, por exemplo. No caso deste exemplo, um comportamento específico para recepção de mensagens fica aguardando ser ativado quando uma nova mensagem chegar ao agente.

Um objeto da classe `Behaviour` também pode se bloquear durante uma certa quantidade de tempo, definida na execução do método `block()`. É importante enfatizar que este método não é igual ao `sleep()` de uma *thread*. O método `block()` não pára a execução de um comportamento no momento em que é invocado. Ele aguarda a finalização do método `action()`. Caso este comportamento ainda não esteja finalizado (`done()` retorna `false`), este é colocado na fila de comportamentos bloqueados. Como já foi dito, a permanência deste comportamento na fila de comportamentos bloqueados pode ser por um determinado tempo, ou até que um evento ocorra.

Como exemplo de utilização do `block()` considere um agente que imprime na tela seu nome local por 10 vezes, sendo que ele possui um intervalo entre as impressões na tela de

5 segundos. O código deste agente está contido nas Caixas de Código 2.6 e 2.7.

Código 2.6: AgenteImpressor.java

```
import jade.core.Agent;
public class AgenteImpressor extends Agent {

    protected void setup() {

        System.out.println("Olá! Eu sou um agente impressor!");
        System.out.println("# Vou executar meu comportamento");
        addBehaviour(new ImprimeFrase(this, 5000));

    }
}
```

Código 2.7: ImprimeFrase.java

```
import jade.core.Agent;
import jade.core.behaviours.Behaviour;

public class ImprimeFrase  extends Behaviour{
    int numExecução=1;
    long delay;
    long tempoInicial = System.currentTimeMillis();

public ImprimeFrase(Agent a, long delay) {
     super(a);
     this.delay = delay;
}

    public void action() {
        block(delay);
        System.out.println("# Tempo " + (System.currentTimeMillis() -
            tempoInicial) + ": Meu nome é " + myAgent.getLocalName());
        numExecução = numExecução+1;
    }

    public boolean done() {
        return numExecução>10;
    }


    public int onEnd() {

       System.out.println(myAgent.getLocalName() + ": Meu comportamento foi
            finalizado! Até mais...");
```

```
        return 0;
        }}
```

Na classe `AgenteImpressor` temos um agente que executa seu comportamento que está implementado na classe `ImprimeFrase`.

A linha de código `addBehaviour(new ImprimeFrase(this,5000))` passa os parâmetros necessários para execução do comportamento. O método `block` recebe valores do tipo `long`, representados em milisegundos. Isto justifica a utilização do valor `5000`, para indicar que o tempo entre cada frase impressa será de 5 segundos.

Observe que o comando `block()` está no início do bloco de comandos do método `action()` da classe `ImprimeFrase`. Isto não indica que o `block()` será executado primeiro. Este será executado ao fim do bloco de comandos do método `action()`.

A classe `ImprimeFrase` também nos revela um novo método: trata-se do método `onEnd()`. Este método retorna um valor inteiro que representa um valor de finalização para o comportamento. Este método é invocado após o comportamento estar concluído e após este ser movido para a lista de comportamentos bloqueados, pois é possível reiniciar este comportamento com a utilização do método `reset()`. É possível, também, utilizar o método `onStart()` que é acionado no início da execução do comportamento.

Com a execução do agente `teste` temos a seguinte saída:

```
java jade.Boot teste:AgenteImpressor
...

Olá! Eu sou um agente impressor!
# Vou executar meu comportamento
# Tempo 0: Meu nome é teste
# Tempo 5015: Meu nome é teste
# Tempo 10031: Meu nome é teste
# Tempo 15046: Meu nome é teste
# Tempo 20062: Meu nome é teste
# Tempo 25078: Meu nome é teste
# Tempo 30093: Meu nome é teste
# Tempo 35109: Meu nome é teste
# Tempo 40125: Meu nome é teste
# Tempo 45140: Meu nome é teste
teste: Meu comportamento foi finalizado! Até mais...
```

Vamos aproveitar este momento para abordar o conceito de concorrência entre agentes. Cada agente JADE é uma *thread*. Isto significa que a execução destes agentes será escalonada. Vamos testar a seguinte aplicação, iniciando três agentes ao mesmo tempo na

plataforma. Cada agente passará como argumento seu tempo de impressão de mensagem na tela. O código referente à classe de agente que vamos utilizar está na Caixa de Código 2.8. O comportamento dos agentes será o mesmo descrito na Caixa de Código 2.7.

Código 2.8: AgenteImpressorArgs.java

```
import jade.core.Agent;
import jade.core.*;

public class AgenteImpressorArgs extends Agent {

    protected void setup() {

        Object[] args = getArguments();
        if(args != null && args.length>0)
        {
        long valor  = Long.parseLong((String) args[0]);
        System.out.println("Olá! Eu sou um agente impressor!");
        System.out.println("# Vou executar meu comportamento");
         addBehaviour(new ImprimeFrase(this,valor));
              }else
        System.out.println("Você não passou argumentos");


    }
}
```

A linha de execução no *prompt* será a seguinte:

```
java jade.Boot Andre:AgenteImpressorArgs(200)
 Maria:AgenteImpressorArgs(400) Paulo:AgenteImpressorArgs(600)
```

Isto implica que o agente `Andre` executará seu comportamento a cada 0.2 seg., o agente `Maria` a cada 0,4 seg. e o agente `Paulo` a cada 0.6 seg. O resultado obtido é o seguinte:

```
Olá! Eu sou um agente impressor!
# Vou executar meu comportamento
# Tempo 0: Meu nome é Andre
Olá! Eu sou um agente impressor!
# Vou executar meu comportamento
# Tempo 0: Meu nome é Maria
Olá! Eu sou um agente impressor!
# Vou executar meu comportamento
# Tempo 0: Meu nome é Paulo
# Tempo 203: Meu nome é Andre
```

```
# Tempo 406: Meu nome é Maria
# Tempo 406: Meu nome é Andre
# Tempo 610: Meu nome é Paulo
# Tempo 610: Meu nome é Andre
# Tempo 813: Meu nome é Maria
# Tempo 813: Meu nome é Andre
# Tempo 1016: Meu nome é Andre
# Tempo 1219: Meu nome é Paulo
# Tempo 1219: Meu nome é Maria
# Tempo 1219: Meu nome é Andre
# Tempo 1422: Meu nome é Andre
# Tempo 1625: Meu nome é Maria
# Tempo 1625: Meu nome é Andre
# Tempo 1828: Meu nome é Paulo
# Tempo 1828: Meu nome é Andre
Andre: Meu comportamento foi finalizado! Até mais...
# Tempo 2031: Meu nome é Maria
# Tempo 2438: Meu nome é Paulo
# Tempo 2438: Meu nome é Maria
# Tempo 2844: Meu nome é Maria
# Tempo 3047: Meu nome é Paulo
# Tempo 3250: Meu nome é Maria
# Tempo 3657: Meu nome é Paulo
# Tempo 3656: Meu nome é Maria
Maria: Meu comportamento foi finalizado! Até mais...
# Tempo 4266: Meu nome é Paulo
# Tempo 4875: Meu nome é Paulo
# Tempo 5485: Meu nome é Paulo
Paulo: Meu comportamento foi finalizado! Até mais...
```

2.5 Comportamentos Pré-Definidos

JADE conta com uma série de comportamentos pré-definidos que auxiliam o desenvolvedor na construção de sistemas multiagentes. Pode-se agrupar os comportamentos oferecidos por JADE em quatro grupos:

1. Comportamentos *one-shot*: tipos de comportamentos que se executam de maneira quase instantânea, e apenas uma vez;

2. Comportamentos cíclicos: são aqueles comportamentos que nunca finalizam. O

método `action()` deste comportamento é sempre executado pois `done()` sempre retorna `false`;

3. Comportamentos temporais: são comportamentos que incluem uma relação temporal em sua execução;

4. Comportamentos compostos: são comportamentos que modelam situações específicas, tais como comportamentos seqüenciais, paralelos, etc.

A seguir tem-se uma breve introdução aos tipos de comportamentos pré-definidos. Os exemplos mais completos destes comportamentos são mostrados no decorrer deste manual, juntamente com o avanço das funcionalidades dos agentes e dos serviços oferecidos pela plataforma.

2.5.1 Comportamentos *One-shot*

Neste comportamento o método `done()` sempre retorna o valor `true`, fazendo com que seja executado apenas uma vez. Para utilização deste comportamento deve-se importar a classe `jade.core.behaviours.OneShotBehaviour`. A estrutura da implementação deste comportamento dá-se conforme ilustrado na Caixa de Código 2.9. Em seguida, na Figura 2.10, ilustra-se o código de agente invocando este comportamento.

Código 2.9: Exemplo *OneShot Behaviour*

```
import jade.core.behaviours.OneShotBehaviour;
public class ComportamentoOneShot extends OneShotBehaviour {
    public void action() {
        //código a ser executado
    }
}
```

Código 2.10: AgenteComportamentoOneShot.java

```
import jade.core.Agent;
import jade.core.behaviours.*;

public class AgenteComportamentoOneShot extends Agent{
    protected void setup() {
        addBehaviour(new ComportamentoOneShot(this));
    }
}
```

2.5.2 Comportamentos Cíclicos

Neste comportamento o método `done()` sempre devolve `false`. Este comportamento se mantém ativo enquanto o agente estiver ativo na plataforma. Para utilização deste comportamento deve-se importar a classe `jade.core.behaviours.CyclicBehaviour`. Sua estrutura de implementação está ilustrada na Caixa de Código 2.11. Em seguida, na Figura 2.12, ilustra-se o código de agente invocando este comportamento.

Código 2.11: Exemplo *Cyclic Behaviour*

```
import jade.core.behaviours.CyclicBehaviour;
public class ComportamentoCiclico extends CyclicBehaviour {
    public void action() {
        //código a ser executado
    }
}
```

Código 2.12: AgenteComportamentoCiclico.java

```
import jade.core.Agent;
import jade.core.behaviours.*;

public class AgenteComportamentoCiclico extends Agent{
    protected void setup() {
        addBehaviour(new ComportamentoCiclico(this, 300));
    }
}
```

2.5.3 Comportamentos Temporais

Neste tipo de comportamentos encontram-se os comportamentos *WakerBehaviour* e *TickerBehaviour*. Ambos possuem uma estreita relação com o tempo durante sua execução. Aos serem invocados, ambos aguardam até que se tenha cumprido um tempo definido (*time-out*) para serem executados. A diferença é que o *WakerBehaviour* executa apenas uma única vez, enquanto que o *TickerBehaviour* realiza um comportamento cíclico.

Como exemplo de um comportamento *WakerBehaviour*, considere um agente que executa seu comportamento após 1 minuto da invocação do mesmo e depois não é mais executado, conforme consta na Caixa de Código 2.13. Para utilização deste comportamento deve-se importar a classe `jade.core.behaviours.WakerBehaviour`. O método que executa a ação do agente do tipo *WakerBehaviour* é o método `onWake()`[2]. Este método é executado logo após o *time-out*.

[2]Pode-se também utilizar o método `void handleElapsedTimeout()`, cuja funcionalidade é a mesma do `onWake()`.

Código 2.13: Exemplo *Waker Behaviour*

```
import jade.core.Agent;
import jade.core.behaviours.WakerBehaviour;
public class Waker extends Agent {
    protected void setup() {
    System.out.println("Adicionando waker behaviour");

        addBehaviour(new WakerBehaviour(this, 10000) {
                protected void onWake() {
                        //realiza operação X
                }
    });
   }
}
```

O *TickerBehaviour* possui seus métodos `action()` e `done()` pré-implementados, bastando para o desenvolvedor implementar o método `onTick()`. Um comportamento *ticker* nunca termina ao menos que seja removido pelos métodos `removeBehaviour()` ou `stop()`.

A Caixa de Código 2.14 ilustra um agente que executa um *TickerBehaviour*, exibindo a cada segundo o número de seu ciclo (*tick*). Para utilização deste comportamento deve-se importar a classe `jade.core.behaviours.TickerBehaviour`. É possível obter o número de ciclos através do método `getTickCount()`, que retorna um valor inteiro que representa o ciclo do agente. Observe que após 5 ciclos o comportamento é interrompido com o método `stop()`. Este também pode ser reiniciado com o o método `reset()`, fazendo com que o comportamento fosse novamente iniciado e o número de ciclos executados seja zerado.

Código 2.14: Exemplo *Ticker Behaviour*

```
import jade.core.Agent;
import jade.core.behaviours.TickerBehaviour;
public class Ticker extends Agent {
     protected void setup() {
    System.out.println("Adicionando TickerBehaviour");

    addBehaviour(new TickerBehaviour(this,1000) {
                protected void onTick() {
                  if(getTickCount()>5){
                        stop();
                    }else
                    //getTickCount() retorna o número de execuções
                    //do comportamento.
                    System.out.println("Estou realizando meu " +
                        getTickCount() + " tick");
                 }
    });
```

```
    }
}
```

Tem-se como resultado da execução deste agente:

```
Adicionando TickerBehaviour
Estou realizando meu 1 tick
Estou realizando meu 2 tick
Estou realizando meu 3 tick
Estou realizando meu 4 tick
Estou realizando meu 5 tick
```

2.5.4 Comportamentos Compostos

Comportamentos compostos são aqueles formados por sub-comportamentos. A política de seleção de comportamentos filhos está implementada em três subclasses: *SequencialBehaviour*, *ParallelBehaviour* e *FSMBehaviour*. Os três tipos de comportamentos compostos serão apresentados nas próximas seções.

2.5.4.1 *SequencialBehaviour*

Este comportamento executa seus sub-comportamentos de maneira seqüencial e termina quando todos seus sub-comportamentos estiverem concluídos. A política de escalonamento dos sub-comportamentos está descrita a seguir. O comportamento inicia executando seu primeiro sub-comportamento. Quando este sub-comportamento é finalizado (isto é, seu método `done()` retorna `true`), o segundo sub-comportamento é executado e assim por diante. Quando o último sub-comportamento é finalizado, o comportamento composto seqüencial é finalizado.

Os sub-comportamentos são adicionados ao comportamento composto com o método `addSubBehaviour()`. A ordem em que estes sub-comportamentos são adicionados indica a ordem de execução dos mesmos. Na Caixa de Código 2.15 tem-se um exemplo de um agente que executa seqüencialmente três comportamentos.

Código 2.15: AgenteSequencial.java

```
import jade.core.Agent;
import jade.core.behaviours.*;;

public class AgenteSequencial extends Agent{

    protected void setup() {
```

```
        //mensagem de inicialização do agente
        System.out.println("Olá! Meu nome é " + getLocalName());
        System.out.println("Vou executar três comportamentos:");

        //criamos um objeto da classe SequentialBehaviour
SequentialBehaviour comportamento = new SequentialBehaviour(this) {

                public int onEnd() {
                myAgent.doDelete();
                return 0;
            }

 };
 //adicionamos seu primeiro comportamento
        comportamento.addSubBehaviour(new WakerBehaviour(this, 500) {
            long t0 = System.currentTimeMillis();
            protected void onWake() {
                System.out.println((System.currentTimeMillis() - t0) +":
                    Executei meu primeiro comportamento após meio segundo!")
                    ;
            }
        });

 //adicionamos seu segundo comportamento
        comportamento.addSubBehaviour(new OneShotBehaviour(this) {

                 public void action() {
                System.out.println("Executei meu segundo comportamento");
            }
        });

//adicionamos seu terceiro comportamento
        comportamento.addSubBehaviour(new TickerBehaviour(this, 700) {
              int exec=0;
              long t1 = System.currentTimeMillis();
               protected void onTick() {
                   if(exec==3) stop();
                   else {
                       System.out.println((System.currentTimeMillis()-t1)+
                           ": Estou executando meu terceiro comportamento");
                       exec++;
                   }
               }
        });

        //acionamos sua execução;
       addBehaviour(comportamento);
```

```
51
    }

54
    protected void takeDown() {
        System.out.println("Fui finalizado com sucesso");
57
    }
}
```

Este agente adiciona três sub-comportamentos para seu comportamento composto. Podemos ter estes três comportamentos em classes separadas e apenas adicioná-las ao comportamento composto.

Executando no *prompt* a linha `java jade.Boot Nick:AgenteSequencial`, tem-se a seguinte execução:

```
Olá! Meu nome é Nick
Vou executar três comportamentos:
515: Executei meu primeiro comportamento após meio segundo!
Executei meu segundo comportamento
1218: Estou executando meu terceiro comportamento
1921: Estou executando meu terceiro comportamento
2625: Estou executando meu terceiro comportamento
Fui finalizado com sucesso
```

2.5.4.2 *ParallelBehaviour*

O *ParallelBehaviours* implementa um comportamento composto que escalona seus sub-comportamentos em paralelo. Toda vez que o método `action()` do comportamento paralelo é executado, o método `action()` de seus sub-comportamentos é executado.

Um comportamento paralelo pode ser instruído para ser finalizado quando todos os comportamentos paralelos estiverem completos, ou quando algum deles é finalizado. Além disto, é possível definir a finalização do comportamento composto para quando um certo número `n` de sub-comportamentos estiverem finalizados. 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 sub-comportamentos que são necessários para finalizar o *ParallelBehaviour*.

A seguir tem-se um exemplo de um agente com comportamento paralelo. O código deste agente está na Caixa de Código 2.16.

Código 2.16: AgenteCompParalelo.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
```

```
3
public class AgenteCompParalelo extends Agent{
6
protected void setup() {
   System.out.println("Olá! Eu sou o agente " + getLocalName());
9  System.out.println("Vou executar três comportamentos concorrentemente");

ParallelBehaviour s = new ParallelBehaviour(this,WHEN_ALL) {
12
            public int onEnd() {
                System.out.println("Comportamento Composto Finalizado com
                    Sucesso!");
15              return 0;
            }

18 };
addBehaviour(s);
s.addSubBehaviour(new SimpleBehaviour(this) {
21      int qtd=1;

            public void action() {
24                  System.out.println("Comportamento 1: Executando pela "
                        + qtd + " vez");
            qtd = qtd+1;

27          }


30          public boolean done() {
                    if(qtd==4) {
                    System.out.println("Comportamento 1 - Finalizado");
33                  return true;
                }else
                    return false;
36              }
            });

39          s.addSubBehaviour(new SimpleBehaviour(this) {
        int qtd=1;

42          public void action() {
                    System.out.println("Comportamento 2: Executando pela "
                        + qtd + " vez");
            qtd = qtd+1;
45
            }
```

```
48
            public boolean done() {
                    if(qtd==8) {
51                  System.out.println("Comportamento 2 - Finalizado");
                    return true;
                }else
54                  return false;
                }
            });
57
            s.addSubBehaviour(new SimpleBehaviour(this) {
        int qtd=1;
60
            public void action() {
                    System.out.println("Comportamento 3: Executando pela "
                       + qtd + " vez");
63          qtd = qtd+1;

            }
66

            public boolean done() {
69                  if(qtd==10) {
                    System.out.println("Comportamento 3 - Finalizado");
                    return true;
72              }else
                    return false;
                }
75          });

            }
78  }
```

Executando a linha de comando `java jade.Boot Azul:AgenteCompParalelo` tem-se o seguinte resultado:

```
Olá! Eu sou o agente Azul
Vou executar três comportamentos concorrentemente
Comportamento 1: Executando pela 1 vez
Comportamento 2: Executando pela 1 vez
Comportamento 3: Executando pela 1 vez
Comportamento 1: Executando pela 2 vez
Comportamento 2: Executando pela 2 vez
Comportamento 3: Executando pela 2 vez
Comportamento 1: Executando pela 3 vez
```

```
Comportamento 1 - Finalizado
Comportamento 3: Executando pela 3 vez
Comportamento 2: Executando pela 3 vez
Comportamento 3: Executando pela 4 vez
Comportamento 2: Executando pela 4 vez
Comportamento 3: Executando pela 5 vez
Comportamento 2: Executando pela 5 vez
Comportamento 3: Executando pela 6 vez
Comportamento 2: Executando pela 6 vez
Comportamento 3: Executando pela 7 vez
Comportamento 2: Executando pela 7 vez
Comportamento 2 - Finalizado
Comportamento 3: Executando pela 8 vez
Comportamento 3: Executando pela 9 vez
Comportamento 3 - Finalizado
Comportamento Composto Finalizado com Sucesso!
```

Observe na linha 11 que o comando `ParallelBehaviour s = new ParallelBehaviour (this, ParallelBehaviour.WHEN_ALL)` constrói o objeto `s` da classe `ParallelBehaviour`. Em seu construtor passamos o parâmetro inteiro `ParallelBehaviour.WHEN_ALL`. Este parâmetro indica que após todos os sub-comportamentos estarem concluídos, o comportamento composto será finalizado, ativando seu método `onEnd()`.

Vamos mudar este parâmetro para `ParallelBehaviour.WHEN_ANY`. Isto indica que quando qualquer um dos sub-comportamentos for finalizado, o comportamento composto é finalizado. Executando novamente o agente obtemos o seguinte resultado:

```
Olá! Eu sou o agente Azul
Vou executar três comportamentos concorrentemente
Comportamento 1: Executando pela 1 vez
Comportamento 2: Executando pela 1 vez
Comportamento 3: Executando pela 1 vez
Comportamento 1: Executando pela 2 vez
Comportamento 2: Executando pela 2 vez
Comportamento 3: Executando pela 2 vez
Comportamento 1: Executando pela 3 vez
Comportamento 1 - Finalizado
Comportamento Composto Finalizado com Sucesso!
```

Como previámos, o comportamento composto foi finalizado após a finalização de um dos sub-comportamentos. Podemos também definir que o comportamento composto finalize após um certo número de finalizações de seus sub-comportamentos. Podemos

utilizar como parâmetro um valor inteiro, por exemplo: `ParallelBehaviour s = new ParallelBehaviour(this, 2)`, onde 2 indica que após dois sub-comportamentos serem finalizados o comportamento composto será finalizado. O resultado da execução do agente com esse novo parâmetro de comportamento é exibido a seguir:

```
Olá! Eu sou o agente Azul
Vou executar três comportamentos concorrentemente
Comportamento 1: Executando pela 1 vez
Comportamento 2: Executando pela 1 vez
Comportamento 3: Executando pela 1 vez
Comportamento 1: Executando pela 2 vez
Comportamento 2: Executando pela 2 vez
Comportamento 3: Executando pela 2 vez
Comportamento 1: Executando pela 3 vez
Comportamento 1 - Finalizado
Comportamento 3: Executando pela 3 vez
Comportamento 2: Executando pela 3 vez
Comportamento 3: Executando pela 4 vez
Comportamento 2: Executando pela 4 vez
Comportamento 3: Executando pela 5 vez
Comportamento 2: Executando pela 5 vez
Comportamento 3: Executando pela 6 vez
Comportamento 2: Executando pela 6 vez
Comportamento 3: Executando pela 7 vez
Comportamento 2: Executando pela 7 vez
Comportamento 2 - Finalizado
Comportamento Composto Finalizado com Sucesso!
```

2.5.4.3 *FSMBehaviour*

Este comportamento é baseado no escalonamento por uma máquina finita de estados (*Finite State Machine*). O *FSMBehaviour* executa cada sub-comportamento de acordo com uma máquina de estados finitos definido pelo usuário. Mais especificamente, cada sub-comportamento representa um estado definido na máquina de estados finitos. Ela fornece métodos para registrar estados (sub-comportamentos) e transições que definem como dar-se-á o escalonamento dos sub-comportamentos. Os passos básicos para se definir um *FSMBehaviour* são:

1. Registrar um comportamento único como estado inicial, passando como parâmetros o comportamento e uma *String* que nomeia este estado. Para isto utiliza-se o método

`registerFirstState();`

2. Registrar um ou mais comportamentos como estados finais, utilizando o método `registerLastState();`

3. Registrar um ou mais comportamentos como estados intermediários utilizando o método `registerState();`

4. Para cada estado, registrar as transições deste com os outros estados utilizando o método `registerTransition()`. Por exemplo, suponha que você tenha um estado definido como `X` e outro estado definido como `Y` e você deseja informar que a transição será feita do estado X para o Y, quando o estado X retornar o valor 1. O método seria definido como `registerTransition(X,Y,1)`.

Considere um agente que realiza um determinado comportamento X. Ao final deste comportamento é verificada se sua operação foi concluída. Caso esta operação não seja concluída, um comportamento Z é efetuado. Ao término do comportamento Z, o comportamento X é executado novamente e toda a verificação ocorre novamente. Na execução em que o comportamento X estiver concluído, será invocado o último comportamento do agente, o comportamento Y. Este algoritmo está ilustrado na Figura 2.2 e o código deste agente encontra-se Caixa de Código 2.17.

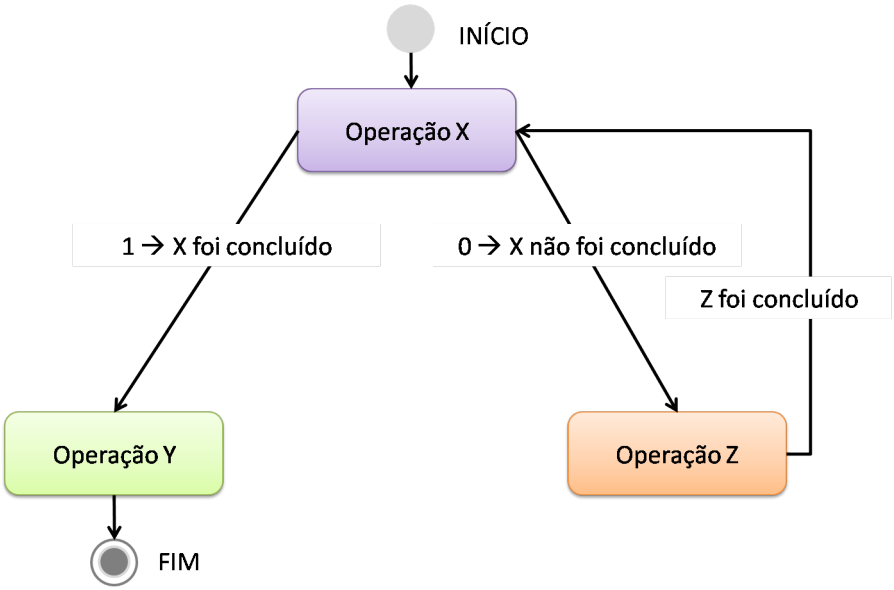

Figura 2.2: Exemplo de Máquina de Estado Finito.

Código 2.17: AgenteFSM.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
public class AgenteFSM extends Agent{

    protected void setup() {

        FSMBehaviour compFSM = new FSMBehaviour(this){

            public int onEnd() {
                System.out.println("Comportamento FSM finalizado com
                    sucesso!");
                return 0;
            }

        };
        //registramos o primeiro comportamento - X
      compFSM.registerFirstState(new OneShotBehaviour(this) {

            int c=0;
            public void action() {
            System.out.println("Executando Comportamento X");
            c++;
            }


            public int onEnd() {
            return (c>4? 1:0);
            }
        }, "X");

        //registramos outro estado - Z
        compFSM.registerState(new OneShotBehaviour(this) {


            public void action() {
                 System.out.println("Executando Comportamento Z");
           }
               public int onEnd() {
            return 2;
            }

        }, "Z");

        //registramos o último estado - Y
        compFSM.registerLastState(new OneShotBehaviour(this) {

```

```
            public void action() {
              System.out.println("Executando meu último comportamento.");
            }
        }, "Y");

        //definimos as transições
        compFSM.registerTransition("X", "Z", 0);  //X --> Z, caso onEnd()
            do X retorne 0
        compFSM.registerTransition("X", "Y", 1);  //X --> Y, caso onEnd()
            do X retorne 1

        //definimos uma transição padrão (não importa tipo de retorno)
        //como a máquina é finita, temos que zerar os estados X e Z --> new
             String[]{"X","Z"}
        compFSM.registerDefaultTransition("Z", "X", new String[]{"X","Z"});
       //Podemos também registrar uma transição quando o estado Z retornar
           2
       //compFSM.registerTransition("Z", "X", 2);

        //acionamos o comportamento
        addBehaviour(compFSM);

    }
    }
```

Ao executarmos `java jade.Boot Caio:AgenteFSM` obtemos o seguinte resultado:

```
Executando Comportamento X
Executando Comportamento Z
Executando Comportamento X
Executando Comportamento Z
Executando Comportamento X
Executando Comportamento Z
Executando Comportamento X
Executando Comportamento Z
Executando Comportamento X
Executando meu último comportamento.
Comportamento FSM finalizado com sucesso!
```

Observe que o comportamento X foi executado 5 vezes e ao final de cada execução o comportamento Z era invocado. Quando `c>4`, a execução de X estava completa e com isto, o último comportamento foi invocado. Observe no código deste agente que definimos um estado com o método `registerState(Comportamento, Nome do Comportamento)`. As transições definimos com `registerTransition(Origem, Destino, Retorno da Origem)`. O método `registerDefaultTransition()` define uma transição padrão, ou seja, uma

transição que ocorre sempre de um estado para o outro independente do retorno obtido na execução do estado de origem.

É importante enfatizar que a linha de código `compFSM.registerDefaultTransition("Z", "X", new String[]{"X", "Z"})` registra uma transição padrão entre Z e X, mas como ambos estados já foram executados e como são comportamentos *one-shot* só poderiam executar uma vez. Por isto, temos o argumento `new String[]{"X", "Z"}` zerando as informações sobre estes estados, possibilitando que possam ser novamente executados.

Capítulo 3

Comunicação entre Agentes

A comunicação entre agentes é fundamental para a execução de sistemas multiagentes. Ela determina o comportamento em uma sociedade, permitindo que um agente não seja apenas um programa que executa seus comportamentos, mas também um programa que recebe e envia pedidos aos demais agentes.

3.1 Envio e Recebimento de Mensagens

A troca de mensagens na plataforma JADE realiza-se mediante mensagens FIPA-ACL. JADE disponibiliza um mecanismo assíncrono de mensagens: cada agente possui uma fila de mensagens (caixa de entrada), onde este agente decide o momento de ler estas mensagens. No momento desejado pelo agente este pode ler apenas a primeira mensagem, ou ler as mensagens que satisfazem algum critério.

As mensagens trocadas são instanciadas da classe `jade.lang.acl.ACLMessage`. A seguir apresenta-se como os agentes podem enviar e receber estas mensagens.

3.1.1 Classe `ACLMessage`

A classe `ACLMessage` disponibiliza uma série de métodos para manipulação das mensagens. Dentre os métodos mais utilizados cita-se:

- `setPerformative(int)`: recebe como parâmetro uma constante que representa o ato comunicativo da mensagem. Por exemplo, uma mensagem `msg` que tenha o ato AGREE será escrita como: `msg.setPerformative(ACLMessage.AGREE)`. Os códigos das *performatives* estão contidos na Tabela 3.1;
- `getPerformative()`: devolve um inteiro equivalente à constante que representa o ato comunicativo da mensagem;

Tabela 3.1: Valores Inteiros das *Performatives*.

INT	*Performative*	INT	*Performative*
0	ACCEPT_PROPOSAL	11	PROPOSE
1	AGREE	12	QUERY-IF
2	CANCEL	13	QUERY-REF
3	CFP	14	REFUSE
4	CONFIRM	15	REJECT_PROPOSAL
5	DISCONFIRM	16	REQUEST
6	FAILURE	17	REQUEST_WHEN
7	INFORM	18	REQUEST_WHENEVER
8	INFORM-IF	19	SUBSCRIBE
9	INFORM-REF	20	PROXY
10	NOT_UNDERSTOOD	21	PROPAGATE

- `createReply()`: cria uma mensagem de resposta para uma mensagem recebida, capturando automaticamente alguns campos tais como: RECEIVER, CONVERSATION-ID, etc;
- `addReceiver(AID)`: recebe como parâmetro um objeto AID (*Agent Identifier*) e o adiciona à lista de receptores;
- `getAllReceiver()`: devolve um objeto *iterator* contendo a lista de receptores;
- `setContent(String)`: recebe como parâmetro uma *String* e a coloca como conteúdo da mensagem;
- `getContent()`: devolve o conteúdo da mensagem;
- `setContentObject(Serializable s)`: recebe como parâmetro um objeto de uma classe que implementa serialização. Por meio deste método é possível transmitir objetos como conteúdos das mensagens;
- `getContentObject()`: devolve o objeto que está no conteúdo da mensagem.

3.1.2 Enviar uma Mensagem

Existem alguns passos a serem seguidos para envio de mensagens. São eles:

1. Crie um objeto `ACLMessage`;
2. Use os métodos disponíveis para preencher os campos necessários (conteúdo, ontologia, receptor, etc);

3. Invoque o método `send()` da classe `Agent`. Este método recebe como parâmetro um objeto `ACLMessage` e adiciona automaticamente o campo do remetente com a identificação do agente, e envia a mensagem aos destinatários.

Por exemplo, considere que desejamos enviar a seguinte mensagem:

```
(QUERY-IF
    :receiver (set (agent-identifier :name pcseller) )
    :content "(pc-offer (mb 256) (processor celeron) (price ?p))"
    :language Jess
    :ontology PC-ontology
)
```

Esta mensagem de exemplo representa um pedido de um agente para o outro, onde o solicitante deseja obter o preço (que será armazenado na variável P) de um computador com processador *Celeron* e 256 MB de memória. Para o envio desta mensagem escreva o código contido na Caixa de Código 3.1.

Código 3.1: Envio de Mensagem

```
ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
msg.setOntology("PC-ontology");
msg.setLanguage("Jess");
msg.addReceiver(new AID("pcseller", AID.ISLOCALNAME);
msg.setContent("(pc-offer (mb 256) (processor celeron) (price ?p))");
send(msg);
```

Observe que utilizamos o método `addReceiver()` pois podemos adicionar vários receptores. Quando sabemos o nome de um agente e queremos que ele seja o receptor da mensagem devemos criar um objeto da classe `AID`. Um objeto da classe `AID` é criado passando as seguintes informações: `AID agente1 = new AID("Nome do Agente", AID.ISLOCALNAME)`, onde o segundo parâmetro indica que o nome que estamos passando não se trata do nome global do agente, mas sim de seu nome local.

3.1.3 Receber uma Mensagem

Para receber uma mensagem deve-se utilizar o método `receive()` da classe `Agent`. Este método captura a primeira mensagem da fila de mensagens (se não houver mensagens, o método retorna `null`). Na Caixa de Código 3.2 tem-se um agente que imprime todas as mensagens que recebe.

Código 3.2: Receiver.java

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;


public class Receiver extends Agent{

    protected void setup() {
        addBehaviour(new CyclicBehaviour(this) {
                public void action() {
                    ACLMessage msg = receive();
                    if(msg!=null)
                     System.out.println(" - " + myAgent.getLocalName() + "
                         <- " + msg.getContent());
    //interrompe este comportamento até que chegue uma nova mensagem
                     block();
                }
        });
    }

}
```

Observe o uso do comando `block()` sem um argumento de *time-out*. Este método coloca o comportamento na lista de comportamentos bloqueados até que uma nova mensagem chegue ao agente. Se não invocamos este método, o comportamento ficará em um *looping* e usará muito a CPU.

3.1.4 Exemplo de Troca de Mensagens

Considere a situação representada na Figura 3.1. Nela, o agente `Alarmado` nota que está acontecendo um incêndio e avisa ao agente `Bombeiro` para que este tome as providências necessárias.

Vamos desenvolver esta situação problema criando duas classes: a classe `AgenteAlarmado` e a classe `AgenteBombeiro`. O agente `Alarmado` envia uma mensagem informando ao agente `Bombeiro` que está acontecendo um incêndio. Ao receber a mensagem, o agente `Bombeiro` ativa os procedimentos para combate do incêndio. Os códigos estão nas Caixas de Código 3.3 e 3.4.

Código 3.3: AgenteAlarmado.java

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
import jade.lang.acl.ACLMessage;
import jade.core.AID;
public class AgenteAlarmado extends Agent{
```

Figura 3.1: Situação Problema - Incêndio.

```
6     protected void setup() {
       addBehaviour(new OneShotBehaviour(this) {

9                 public void action() {
                  ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                  msg.addReceiver(new AID("Bombeiro",AID.ISLOCALNAME));
12                msg.setLanguage("Português");
                  msg.setOntology("Emergência");
                  msg.setContent("Fogo");
15                myAgent.send(msg);

                  }
18          });
    }
}
```

Código 3.4: AgenteBombeiro.java

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
3 import jade.lang.acl.ACLMessage;

public class AgenteBombeiro extends Agent{
6

    protected void setup() {
        addBehaviour(new CyclicBehaviour(this) {
9

            public void action() {
                ACLMessage msg = myAgent.receive();
12              if(msg != null) {
```

```
                String content = msg.getContent();
            //com equalsIgnoreCase fazemos uma comparação
            //não case-sensitive.
                if(content.equalsIgnoreCase("Fogo")) {
                    System.out.println("O agente " + msg.getSender().
                        getName() +
                            " avisou de um incêndio");
                    System.out.println("Vou ativar os procedimentos de
                        combate ao incêndio!");
                }
            } else
            block();
            } //fim do action()
        }); //fim do addBehaviour()
    }
}
```

Observe na classe `AgenteAlarmado` que escolhemos o receptor da mensagem com a linha de comando: `msg.addReceiver(new AID("Bombeiro",AID.ISLOCALNAME))`. Quando conhecemos o nome de um agente, no caso `Bombeiro`, adicionamos este nome em um objeto AID e informamos com o parâmetro `AID.ISLOCALNAME` que trata-se do nome local do agente.

Execute no *prompt* a linha:

```
java jade.Boot Bombeiro:AgenteBombeiro A1:AgenteAlarmado
```

Obterá o seguinte resultado:

```
O agente A1@lap:1099/JADE avisou de um incêndio
Vou ativar os procedimentos de combate ao incêndio!
```

Vamos agora incrementar nossos agentes. O agente `Alarmado` envia uma mensagem para o agente `Bombeiro`. O agente `Bombeiro` recebe e processa esta mensagem. Agora, o agente `Bombeiro` irá responder ao agente `Alarmado`.

Os códigos desta nova funcionalidade estão nas Caixas de Código 3.5 e 3.6, respectivamente.

Código 3.5: AgenteAlarmado2.java

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
import jade.core.AID;
public class AgenteAlarmado2 extends Agent{
```

```
  protected void setup() {


        addBehaviour(new OneShotBehaviour(this) {

            public void action() {
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(new AID("Bombeiro",AID.ISLOCALNAME));
            msg.setLanguage("Português");
            msg.setOntology("Emergência");
            msg.setContent("Fogo");
            myAgent.send(msg);

            }
        });
        addBehaviour(new CyclicBehaviour(this) {


            public void action() {
            ACLMessage msg = myAgent.receive();
            if(msg != null){
                String content = msg.getContent();
                System.out.println("--> " + msg.getSender().getName() + ":
                    " + content);
            }else
            //Com o block() bloqueamos o comportamento até que uma nova
            //mensagem chegue ao agente e assim evitamos consumir ciclos
            //da CPU.
            block();

            }
        });



}
 }
```

Código 3.6: AgenteBombeiro2.java

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

public class AgenteBombeiro2 extends Agent{

```

```
    protected void setup() {
        addBehaviour(new CyclicBehaviour(this) {

            public void action() {
                ACLMessage msg = myAgent.receive();

                if(msg != null) {
                        ACLMessage reply = msg.createReply();
                    String content = msg.getContent();
                   if(content.equalsIgnoreCase("Fogo")) {
                      reply.setPerformative(ACLMessage.INFORM);
                      reply.setContent("Recebi seu aviso! Obrigado por
                          auxiliar meu serviço");
                      myAgent.send(reply);
                      System.out.println("O agente "+ msg.getSender().
                          getName() +" avisou de um incêndio");
                      System.out.println("Vou ativar os procedimentos de
                          combate ao incêndio!");
                   }
                } else
                  block();
                }
        });
    }
}
```

Observe o uso do método `createReply()` na classe `AgenteBombeiro2`. Este método auxilia na resposta de mensagens recebidas, encapsulando automaticamente no objeto `reply` o endereço do destinatário da resposta.

Vamos executar estes agentes de um modo diferente do que estamos fazendo até agora: vamos executar os agentes a partir de *prompts* diferentes. Para que um agente JADE possa ser criado este deve estar em um *container*. Este *container* deve ser criado no mesmo instante em que o agente é criado. Quando executávamos a linha `java jade.Boot agente:ClassedoAgente agente2:ClassedoAgente` estávamos criando agentes em um *container* nativo, o *Main Container*. Neste caso, vamos iniciar o agente `Bombeiro` no *main container* e criar o agente `Alarmado` em outro *container*. Criamos o agente `Bombeiro` com a seguinte linha:

```
java jade.Boot Bombeiro:AgenteBombeiro2
```

Até agora nada mudou na nossa maneira de iniciar um agente. Como vamos iniciar outro agente, poderíamos pensar em executar novamente a última linha de comando. No entanto, agora precisamos passar o parâmetro `-container` para que o agente Alarmado seja criado. Assim:

```
java jade.Boot -container A1:AgenteAlarmado2
```

Executando esses agentes, obtemos os seguintes resultados:

- No *prompt* do Agente Bombeiro:

```
O agente A1@lap:1099/JADE avisou de um incêndio
Vou ativar os procedimentos de combate ao incêndio!
```

- No *prompt* do Agente Alarmado A1:

```
--> Bombeiro@lap:1099/JADE: Recebi seu aviso!
Obrigado por auxiliar meu serviço
```

Observe que durante o desenvolvimento de nosso agente utilizamos o método `block()` para evitar o consumo da CPU. Existe uma peculiaridade da plataforma JADE que deve ser observada durante o desenvolvimento de um agente. Quando uma mensagem chega a um determinado agente, todos os seus comportamentos que estão bloqueados são desbloqueados. Logo, se um agente possui um comportamento ou mais comportamentos bloqueados, todos serão desbloqueados com a chegada de uma mensagem. Pode-se usar como correção desta peculiaridade condições (*if-elses*) da linguagem JAVA para que um comportamento não seja executado e volte a se bloquear novamente. É importante lembrar que os comportamentos *CyclicBehaviour*, *TickerBehaviour* e *WakerBehaviour* já foram implementados com estas funcionalidades.

3.2 Envio de Objetos

Em muitas situações costuma-se transferir uma grande quantidade de informações. Passar essas informações como *strings* requer um pouco mais de tempo de execução da aplicação, e a implementação pode perder eficiência na transmissão das informações.

A linguagem JAVA possui um recurso que permite o envio de objetos pela *stream* de dados. Este processo é denominado de serialização. Tecnicamente falando, serializar é transformar um objeto em uma seqüência de *bytes*, que poderá ser armazenado em algum local (arquivo de dados, por exemplo) e futuramente ser deserializado para seu estado original.

A plataforma JADE disponibiliza métodos que permitem a transmissão de objetos como conteúdo de mensagens. Estes métodos são `setContentObject(Serializable s)` e `getContentObject()`. Para exemplo de utilização destas funcionalidades considere a

seguinte situação problema. Em uma loja de música temos dois funcionários: o agente `Estoque` e o agente `Contador`. O agente `Estoque` deve informar ao agente `Contador` os músicos contidos no estoque da loja. Para isto, devem ser informados o nome do músico, sua idade e a banda que este compõe, caso participe de uma. Este cenário está ilustrado na Figura 3.2. Nela, o agente `Estoque` envia as informações para o agente `Contador` que lista as informações recebidas.

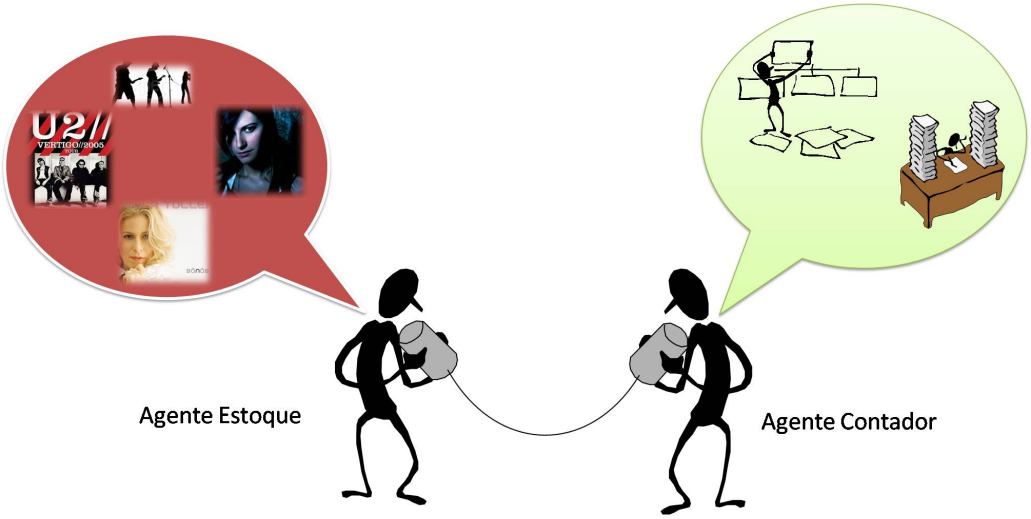

Figura 3.2: Cenário - Loja de Música.

Vamos desenvolver este cenário aplicando as utilidades da serialização de objetos, para que estes objetos possam ser o conteúdo das mensagens. Para informações sobre os músicos temos a classe `Musicos`, cujo código está descrito na Caixa de Código 3.7.

O agente `Estoque` criará cinco objetos da classe `Musicos` contendo as informações sobre os músicos, e esses objetos serão enviados para o agente `Contador`. As funcionalidades do agente `Estoque` estão na Caixa de Código 3.8.

Código 3.7: Musicos.java

```
import jade.util.leap.Serializable;
/* Uma classe que terá seus objetos
 * serializados deve implementar a interface Serializable */

public class Musicos implements Serializable {
    String nome;
    int idade;
    String banda;

```

```
    public Musicos(String nome, int idade, String banda) {
        this.nome = nome;
        this.idade = idade;
        this.banda = banda;
    }

    public void Imprimir() {
        System.out.println("----------------------------");
        System.out.println("Nome...: " + nome);
        System.out.println("Idade...: " + idade);
        System.out.println("Banda...: " + banda);
        System.out.println("----------------------------\n");
    }
}
```

Código 3.8: AgenteEstoque.java

```
import jade.core.Agent;
import jade.core.behaviours.SimpleBehaviour;
import jade.lang.acl.ACLMessage;
import jade.core.AID;
import java.io.IOException;

public class AgenteEstoque extends Agent {

    Musicos[] mus = new Musicos[5];

    protected void setup() {

        mus[0] = new Musicos("Cláudia Leite", 30, "Babado Novo");
        mus[1] = new Musicos("Paula Toller", 45, "Kid Abelha");
        mus[2] = new Musicos("Rogério Flausino", 37, "Jota Quest");
        mus[3] = new Musicos("Laura Pausini", 33, null);
        mus[4] = new Musicos("Bono Vox", 47, "U2");

        addBehaviour(new SimpleBehaviour(this) { //início do comportamento

            int cont = 0;

            public void action() {
                try {
                    ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                    msg.addReceiver(new AID("Contador", AID.ISLOCALNAME));
                    msg.setContentObject(mus[cont]);
                    myAgent.send(msg); //envia a mensagem
                    cont = cont + 1;
                  //  block(100);
                } catch (IOException ex) {
```

```
                    System.out.println("Erro no envio da mensagem");
33              }
            }

36          public boolean done() {

                if (cont > 4) {
39                  myAgent.doDelete(); //finaliza o agente
                    return true;
                } else {
42                  return false;
                }
            }
45      }); //fim do comportamento
    } //fim do método setup() do agente

48  //A invocação do método doDelete() aciona o método takeDown()
    protected void takeDown() {
        System.out.println("Todas as informações foram enviadas");
51  }
}
```

O agente `Contador` receberá as mensagens enviadas pelo agente `Estoque` e os objetos recebidos terão suas informações impressas na tela. As funcionalidades do agente `Contador` estão na Caixa de Código 3.9.

Código 3.9: AgenteContador.java

```
import jade.core.Agent;
3 import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

6 public class AgenteContador extends Agent {

    protected void setup() {
9       System.out.println("Agente Contador inicializado.\n " +
                "Aguardando informações...");

12      addBehaviour(new CyclicBehaviour(this) {  //início do comportamento

            Musicos[] musicos = new Musicos[5]; //vetor da classe Musicos
15          int cont = 0;

            public void action() {
18
                ACLMessage msg = receive(); //captura nova mensagem
```

```
21                  if (msg != null) { //se existe mensagem
                        try { //extrai o objeto
                            musicos[cont] = (Musicos) msg.getContentObject();
24                          //imprime as informações do objeto
                            musicos[cont].Imprimir();
                            cont = cont + 1;
27                      } catch (Exception e) {
                        }
                    } else
30                  block(); //aguarda nova mensagem
                }
            });//término do comportamento
33      } //fim do método setup() do agente
}
```

Executando o agente `Contador` com a seguinte linha:

```
java jade.Boot Contador:AgenteContador
```

Obtemos o seguinte resultado:

```
Agente Contador inicializado.
Aguardando informações...
```

Vamos executar agora o agente `Estoque`, com a seguinte linha:
`java jade.Boot -container Estoque:AgenteEstoque`
Quando o agente `Estoque` é iniciado, este envia as mensagens para o agente `Contador` e após o término do envio obtemos o seguinte resultado:

```
Todas as informações foram enviadas.
```

E no *prompt* do agente `Contador` obtemos o seguinte resultado:

```
----------------------------
Nome...: Cláudia Leite
Idade...: 30
Banda...: Babado Novo
----------------------------

----------------------------
Nome...: Paula Toller
```

```
Idade...: 45
Banda...: Kid Abelha
----------------------------

----------------------------
Nome...: Rogério Flausino
Idade...: 37
Banda...: Jota Quest
----------------------------

----------------------------
Nome...: Laura Pausini
Idade...: 33
Banda...: null
----------------------------

----------------------------
Nome...: Bono Vox
Idade...: 47
Banda...: U2
----------------------------
```

3.3 Seleção de Mensagens

Para selecionar as mensagens que um agente deseja receber podemos utilizar a classe `jade.lang.acl.MessageTemplate`. Esta classe permite definir filtros para cada atributo da mensagem `ACLMessage` e estes filtros podem ser utilizados como parâmetros do método `receive()`.

Nesta classe se define um conjunto de métodos estáticos que retornam como resultado um objeto do tipo `MessageTemplate`. As opções para filtragem das mensagens são:

- `MatchPerformative(`*`performative`*`)`: permite selecionar os tipos de atos comunicativos das mensagens que serão aceitos;
- `MatchSender(AID)`: permite selecionar um ou um grupo de agentes cujas mensagens enviadas por estes serão lidas;
- `MatchConversationID(String)`: permite que apenas mensagens de um determinado tópico sejam lidas. Um tópico nada mais é do que uma *String* definido por um

agente. Todas as mensagens relacionadas a este tópico contêm essa *string*, e serão lidas baseadas neste filtro;

- `and(MessageTemplate1, MessageTemplate2)`: realiza um "E" lógico entre dois filtros;
- `or(MessageTemplate1, MessageTemplate2)`: realiza um "OU" lógico entre dois filtros;
- `not(MessageTemplate)`: inverte o filtro;
- `MatchOntology(String)`: permite que mensagens com uma determinada ontologia sejam lidas;
- `MatchProtocol(String)`: permite que mensagens envolvidas em um protocolo sejam lidas;
- `MatchLanguage(String)`: permite que mensagens em uma certa linguagem sejam lidas;
- `MatchContent(String)`: permite que mensagens com um determinado conteúdo sejam lidas;
- `MatchReplyWith(String)`: permite que um filtro seja realizado de acordo com o campo `replywith`.

Existe também o método `match(ACLMessage)` que retorna um valor booleano verdadeiro caso a mensagem que está em seu parâmetro respeite os filtros definidos pelo objeto `MessageTemplate`.

O nosso agente `Bombeiro`, citado na Seção 3.1.4 (página 35), poderia estar interessado em receber apenas mensagens do tipo INFORM e cuja linguagem seja o `Português`. Seu código ficaria como mostrado na Caixa de Código 3.10.

Código 3.10: AgenteBombeiroFiltro.java

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class AgenteBombeiroFiltro extends Agent {

    protected void setup() {
        addBehaviour(new CyclicBehaviour(this) { //Início do Comportamento

            public void action() {
```

```
12
                    //definimos o primeiro filtro
                    MessageTemplate MT1 = MessageTemplate.MatchPerformative(
                        ACLMessage.INFORM);
15                  //definimos o segundo filtro
                    MessageTemplate MT2 = MessageTemplate.MatchLanguage("
                        Português");
                    //Realizamos um "E" lógico entre os dois filtros
18                  MessageTemplate MT3 = MessageTemplate.and(MT1, MT2);
                    //Recebe a mensagem de acordo com o filtro
                    ACLMessage msg = myAgent.receive(MT3);
21
                    if (msg != null) {
                        String content = msg.getContent();
24                      if (content.equalsIgnoreCase("Fogo")) {
                            System.out.println("O agente " + msg.getSender().
                                getName() + "avisou de um incêndio");
                        }
27
                    }
                }
30          }); //Fim do Comportamento
    }
}
```

3.4 Páginas Amarelas

Já vimos como ocorre a comunicação entre agentes. Agora temos um novo dilema: como um agente pode localizar outros agentes que oferecem um determinado serviço, e obter seus identificadores para que possam se comunicar?

Para que isto possa ser feito, a plataforma JADE implementa o serviço de páginas amarelas em um agente: o agente *Directory Facilitator* (DF), seguindo as especificações do padrão FIPA. Agentes que desejam divulgar seus serviços registram-se no DF, e os demais podem então buscar por agentes que provêm algum serviço desejado.

O DF nada mais é do que um registro centralizado cuja entradas associam a ID do agente aos seus serviços. Para criação e manipulação (busca) dessas entradas utilizamos um objeto `DFAgentDescription`. Para registrar o serviço de um agente devemos fornecer uma descrição deste serviço e a AID do agente. Para uma busca, basta fornecer a descrição do serviço. Essa busca retornará um *array* com ID's dos agentes que oferecem o serviço buscado.

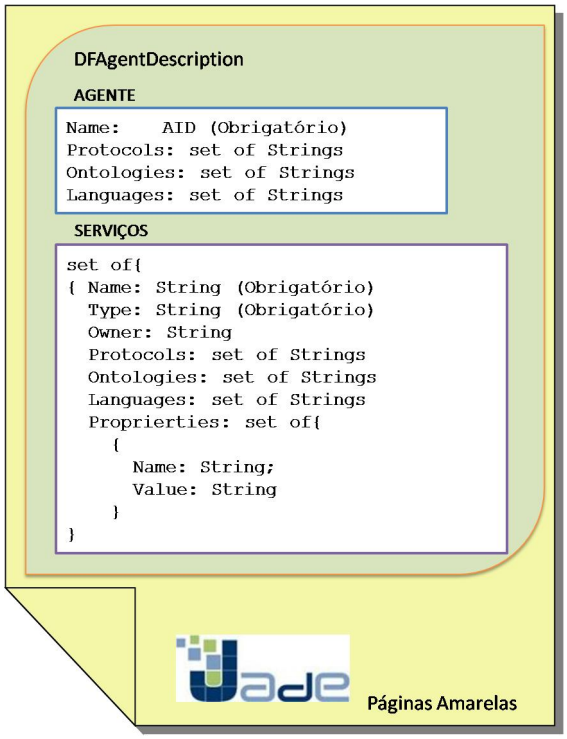

Figura 3.3: Estrutura de uma Entrada no DF.

Na Figura 3.3 tem-se a estrutura de uma entrada no registro do DF. Ela parece ser complexa, mas seu uso se torna simples pois apenas alguns dos campos fornecidos são

utilizados. Para cada serviço publicado devemos fornecer(obrigatoriamente) a AID do agente, o tipo e o nome do serviço. Podemos também fornecer os protocolos, linguagens e ontologias que um agente manipula para que os demais possam conhecer e interagir com este agente.

Para manipulação destes dados, os seguintes métodos estão disponíveis:

- `static DFAgentDescription register`: registra os serviços de um agente no DF;
- `static void deregister`: elimina o registro do DF os serviços fornecidos pelo agente;
- Os serviços são definidos com os métodos da classe `ServiceDescription`:
  - `void setName`: definimos o nome do serviço;
  - `void setOwnership`: definimos o proprietário do serviço;
  - `void setType`: definimos o tipo do serviço. Este valor é definido pelo desenvolvedor, não existem tipos pré-definidos;
  - `void addLanguages`: adicionamos linguagens ao serviço;
  - `void addOntologies`: adicionamos ontologias ao serviço;
  - `void addProtocols`: adicionamos protocolos ao serviço;
  - `void addProperties`: adicionamos propriedades ao serviço. Este valor é definido pelo desenvolvedor, não existem propriedades pré-definidas.
- A descrição do agente que oferece o serviço é realizada com o uso dos métodos da classe `DFAgentDescription`:
  - `void setName`: definimos o AID do agente;
  - `void addServices`: adicionamos o serviço passado como parâmetro deste método à descrição do agente;
  - `void removeServices`: removemos um serviço oferecido da descrição deste agente;
  - `void addLanguages`: adicionamos linguagens que o agente entende;
  - `void addOntologies`: adicionamos ontologias que o agente manipula;
  - `void addProtocols`: adicionamos protocolos que o agente manipula.

3.4.1 Registro

Para que um agente divulgue seus serviços, este deve se registrar nas páginas amarelas da plataforma, isto é, deve se registrar no DF. Costuma-se definir o registro no DF como a primeira ação do agente em seu método `setup()`. Para isto, utilizamos o método `register()` fornecendo como parâmetro um objeto `DFAgentDescription`. Este método deve ser envolvido por um bloco de exceção.

Por exemplo, considere que um agente ofereça um determinado serviço. Este agente poderia se registrar no DF da maneira descrita na Caixa de Código 3.11.

Código 3.11: Registro.java

```
import jade.core.Agent;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;

public class Registro extends Agent {
    protected void setup() {
        //Criamos uma entrada no DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID()); //Informamos a AID do agente

        //Vamos criar um serviço
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Tipo"); //Tipo do Serviço
        sd.setName("Serviço1"); //Nome do Serviço
        //adicionamos o Serviço1
        dfd.addServices(sd);

        //Vamos criar outro serviço
        sd = new ServiceDescription();
        sd.setType("Tipo de Serviço");
        sd.setName("Serviço2");
        dfd.addServices(sd);

        //Vamos registrar o agente no DF
        try {
            //register(agente que oferece, descrição)
            DFService.register(this, dfd);

        } catch (FIPAException e) {
            e.printStackTrace();
        }
    }
}
```

Uma boa prática é remover o registro do agente quando este termina sua execução. Quando um agente é finalizado, automaticamente seu registro é removido das páginas brancas, mas não é removido das páginas amarelas. Por isto, costuma-se remover o registro do agente no método `takeDown()`. O método `takeDown()` fica implementado da seguinte maneira:

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

3.4.2 Busca

Para buscar no DF devemos criar um objeto `DFAgentDescription`, agora sem a AID do agente. O processo é muito semelhante ao de registro, pois criamos a descrição do serviço buscado da mesma maneira que criamos a descrição de um serviço oferecido por um agente. A implementação de um agente que realiza a busca no DF está contida na Caixa de Código 3.12.

Código 3.12: Busca.java

```
import jade.core.Agent;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import java.util.Iterator;

public class Busca extends Agent {

    protected void setup() {

     //crio uma entrada no DF
     DFAgentDescription template = new DFAgentDescription();

     //crio um objeto contendo a descrição do serviço
     ServiceDescription sd = new ServiceDescription();
     sd.setType("Tipo"); //defino o tipo de serviço
     /*Neste momento poderia definir outras características
     * do serviço buscado para filtrar melhor a busca.
     * No caso, vamos buscar por serviços do tipo "Tipo" */

```

```
        //adiciono o serviço na entrada
24      template.addServices(sd);
        try {

27        //Vou buscar pelos agentes
          //A busca retorna um array DFAgentDescription
          //O parâmetro this indica o agente que está realizando a busca
30        DFAgentDescription[] result = DFService.search(this, template);

          //Imprimo os resultados
33        for (int i = 0; i < result.length; i++) {
          //result[i].getName() fornece a AID do agente
          String out = result[i].getName().getLocalName() + " provê ";
36
          //Para obter os serviços do agente invocamos
          //o método getAllServices();
39        Iterator iter = result[i].getAllServices();

          while (iter.hasNext()) {
42              //Extraímos os servicos para um objeto ServiceDescription
                ServiceDescription SD = (ServiceDescription) iter.next();
                //Capturamos o nome do serviço
45              out += " " + SD.getName();
                    }
          //Os serviços de cada agente são impressos na tela
48        System.out.println(out);
                } //fim do laço for


51
           } catch (FIPAException e) {
                e.printStackTrace();
54         }
      }
}
```

3.4.3 Solicitando Ajuda

Considere a seguinte situação: um agente solicitante observa uma situação problema em seu ambiente, tal como um assalto, uma pessoa doente e um incêndio. Este agente então busca por outros agentes que possam resolver estes problemas. No caso um agente policial, um agente médico e um agente bombeiro. Após encontrar estes agentes, o agente solicitante comunica o que está acontecendo. Este cenário está ilustrado na Figura 3.4.

No contexto multiagentes, o agente solicitante busca nas páginas amarelas da plataforma por agentes que ofereçam determinado serviço. Estes agentes devem, ao iniciar, cadastrar seus serviços nas páginas amarelas. Para a implementação deste contexto te-

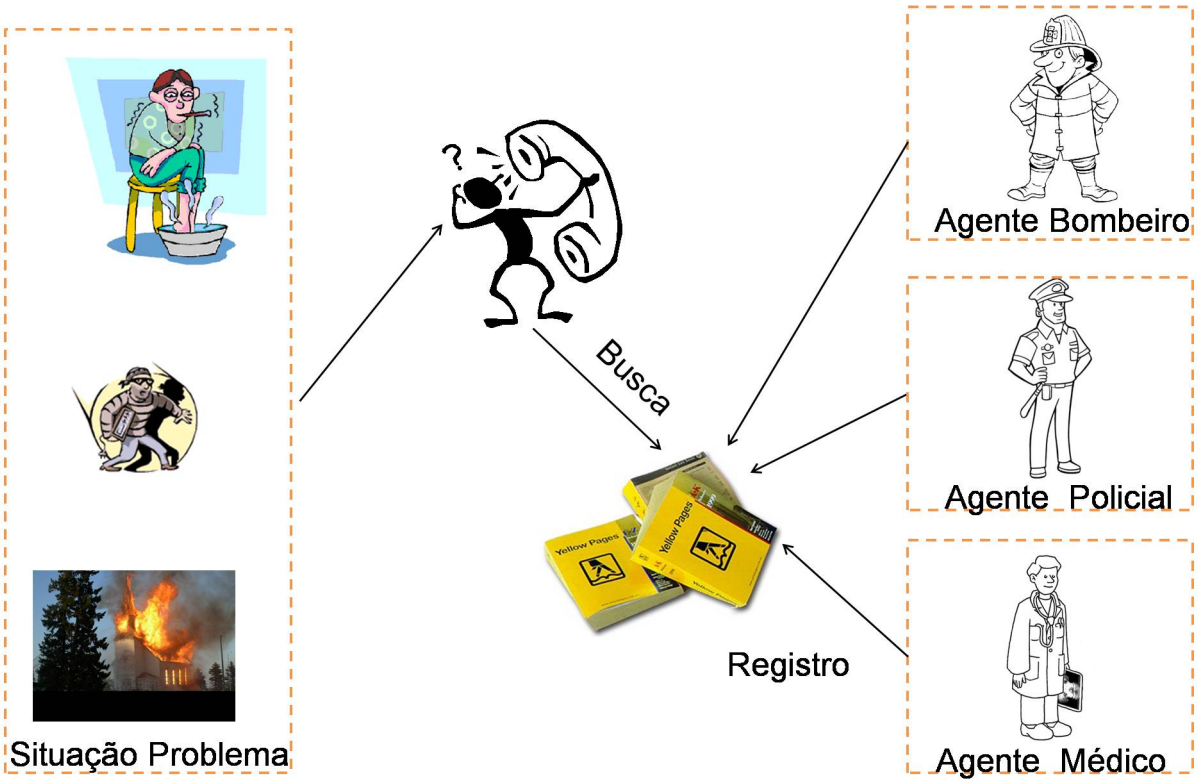

Figura 3.4: Cenário - Solicitando Ajuda.

mos as classes `Solicitante, Bombeiro, Medico` e `Policial`. A primeira refere-se ao agente solicitante e as demais aos agentes prestadores de serviços. Estas classes estão nas Caixas de Código 3.13, 3.14, 3.15 e 3.16, respectivamente.

Código 3.13: Solicitante.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;

public class Solicitante extends Agent {

    protected void setup() {
        //Captura argumentos
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            String argumento = (String) args[0];
```

```
            //Se o argumento é fogo
            if (argumento.equalsIgnoreCase("fogo")) {
                ServiceDescription servico = new ServiceDescription();
                //O serviço é apagar fogo
                servico.setType("apaga fogo");
                //busca por quem fornece o serviço
                busca(servico, "fogo");
            }
            //Se o argumento é ladrão
            if (argumento.equalsIgnoreCase("ladrão")) {
                ServiceDescription servico = new ServiceDescription();
                //O serviço  prender o ladrão
                servico.setType("prende ladrão");
                busca(servico, "ladrão");
            }
            //Se o argumento é doente
            if (argumento.equalsIgnoreCase("doente")) {
                ServiceDescription servico = new ServiceDescription();
                //O serviço é salvar vidas
                servico.setType("salva vidas");
                busca(servico, "doente");
            }
            //Comportamento para receber mensagens
            addBehaviour(new CyclicBehaviour(this) {

                public void action() {
                    ACLMessage msg = receive();
                    if (msg != null) {
                        System.out.println(msg.getSender() + " : " + msg.
                            getContent());
                    }else
                    block();
                }
            });
        }
    }
    //Método que realiza a busca nas páginas amarelas da plataforma
    protected void busca(final ServiceDescription sd, final String Pedido)
        {

        //A cada minuto tenta buscar por agentes que fornecem
        //o serviço
        addBehaviour(new TickerBehaviour(this, 60000) {

            protected void onTick() {
                DFAgentDescription dfd = new DFAgentDescription();
                dfd.addServices(sd);
```

```
                try {
                    DFAgentDescription[] resultado = DFService.search(
                        myAgent, dfd);
                    if (resultado.length != 0) {
                        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                        msg.addReceiver(resultado[0].getName());
                        msg.setContent(Pedido);
                        myAgent.send(msg);
                        stop(); //finaliza comportamento
                    }
                } catch (FIPAException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

Código 3.14: Bombeiro.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;

public class Bombeiro extends Agent {

    protected void setup() {
        //Descrição do Serviço
        ServiceDescription servico = new ServiceDescription();
        //Seu serviço é salvar vidas
        servico.setType("apaga fogo");
        servico.setName(this.getLocalName());
        registraServico(servico);
        RecebeMensagens("fogo", "Vou apagar o incêndio");

    }

    //método para registrar serviço
    protected void registraServico(ServiceDescription sd) {
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
```

```
        } catch (FIPAException e) {
            e.printStackTrace();
        }

    }
    //Método para adicionar um comportamento para receber mensagens
    protected void RecebeMensagens(final String mensagem, final String resp
        ) {
        addBehaviour(new CyclicBehaviour(this) {

            public void action() {
                ACLMessage msg = receive();
                if (msg != null) {
                    if (msg.getContent().equalsIgnoreCase(mensagem)) {
                        ACLMessage reply = msg.createReply();
                        reply.setContent(resp);
                        myAgent.send(reply);
                    }
                }else
                block();
            }
        });
    }
}
```

Código 3.15: Medico.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;

public class Medico extends Agent {

    protected void setup() {
        //Descrição do Serviço
        ServiceDescription servico = new ServiceDescription();
        //Seu serviço é salvar vidas
        servico.setType("salva vidas");
        servico.setName(this.getLocalName());
        registraServico(servico);
        RecebeMensagens("doente", "Vou salvar o doente");

    }
```

```
    //método para registrar serviço
    protected void registraServico(ServiceDescription sd) {
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        } catch (FIPAException e) {
            e.printStackTrace();
        }

    }
    //Método para adicionar um comportamento para receber mensagens
    protected void RecebeMensagens(final String mensagem, final String resp
        ) {
        addBehaviour(new CyclicBehaviour(this) {

            public void action() {
                ACLMessage msg = receive();
                if (msg != null) {
                    if (msg.getContent().equalsIgnoreCase(mensagem)) {
                        ACLMessage reply = msg.createReply();
                        reply.setContent(resp);
                        myAgent.send(reply);
                    }
                }else
                block();
            }
        });
    }
}
```

Código 3.16: Policial.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;

public class Policial extends Agent {

    protected void setup() {
        //Descrição do Serviço
        ServiceDescription servico = new ServiceDescription();
```

```
15          //Seu serviço é salvar vidas
            servico.setType("prende ladrão");
            servico.setName(this.getLocalName());
18          registraServico(servico);
            RecebeMensagens("ladrão", "Vou prender o ladrão");

21      }

        //método para registrar serviço
24      protected void registraServico(ServiceDescription sd) {
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.addServices(sd);
27          try {
                DFService.register(this, dfd);
            } catch (FIPAException e) {
30              e.printStackTrace();
            }

33      }
        //Método para adicionar um comportamento para receber mensagens
        protected void RecebeMensagens(final String mensagem, final String resp
            ) {
36          addBehaviour(new CyclicBehaviour(this) {

                public void action() {
39                  ACLMessage msg = receive();
                    if (msg != null) {
                        if (msg.getContent().equalsIgnoreCase(mensagem)) {
42                          ACLMessage reply = msg.createReply();
                            reply.setContent(resp);
                            myAgent.send(reply);
45                      }
                    }else
                    block();
48              }
            });
        }
51  }
```

Vamos rodar este cenário de uma nova maneira: utilizando ferramentas gráficas da plataforma JADE. A plataforma JADE possui um agente que fornece uma interface gráfica de administração da plataforma. Trata-se do agente RMA (*Remote Management Agent*). Existe um atalho para execução deste agente: basta incluir o parâmetro `-gui` na linha de execução da plataforma (e.g, `java jade.Boot -gui ...`).

Com a invocação deste agente é exibida a interface gráfica ilustrada na Figura 3.5. Automaticamente com o comando `-gui`, os agentes AMS e DF também são carregados.

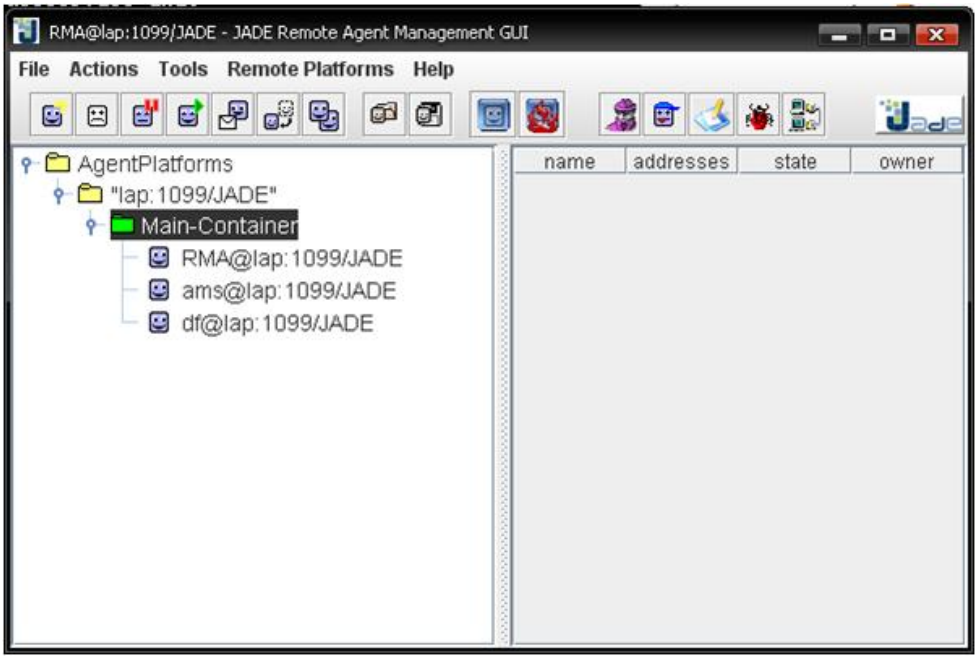

Figura 3.5: Interface Gráfica da Plataforma JADE.

Ao longo deste manual as funcionalidades desta interface serão abordadas. Mas no momento estamos interessados em estudar apenas uma destas funcionalidades: o `Agente Sniffer`. Este agente intercepta mensagens ACL e as mostra graficamente usando uma notação semelhante aos diagramas de seqüência UML. A Figura 3.6 ilustra a atividade deste agente.

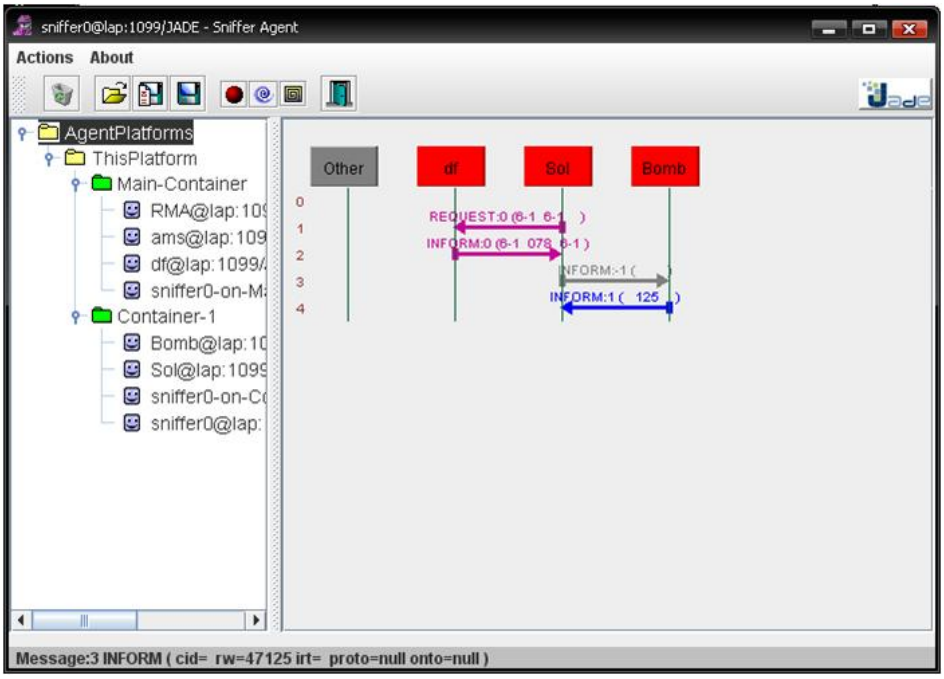

Figura 3.6: Agente Sniffer.

Vamos iniciar nosso agente `Solicitante` com a seguinte linha de comando:

```
java jade.Boot -gui Pedinte:Solicitante(fogo)
```

Com esta execução criamos um agente `Pedinte` que solicita às páginas amarelas um agente que combate o fogo, no caso o `Bombeiro`. A tela gráfica desta execução está ilustrada na Figura 3.7.

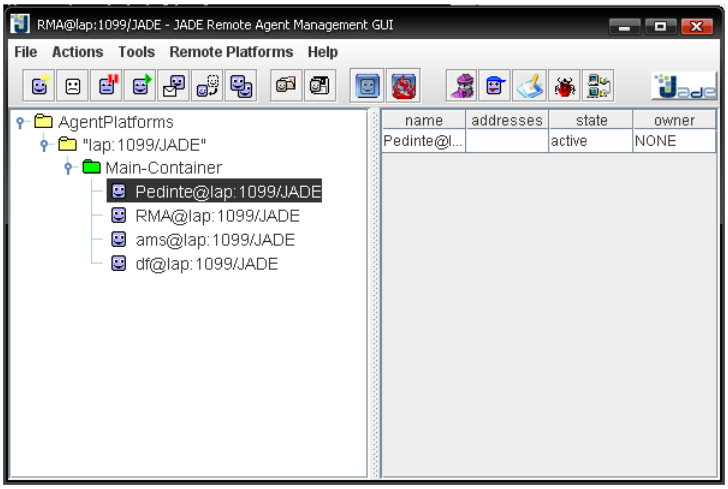

Figura 3.7: Agente Pedinte na Plataforma.

Vamos iniciar o agente `Sniffer`. Clique sobre algum *container* ou sobre alguma plataforma. Agora vamos no menu `Tools -> Start Sniffer`. A tela do `Sniffer` será aberta, onde teremos a lista de agentes na plataforma. Para verificar o fluxo de mensagens entre agentes temos que selecioná-los, e para tanto basta clicar com o botão direito do *mouse* e escolher a opção *Do sniff this agent(s)*. Estes passos estão ilustrados na Figura 3.8.

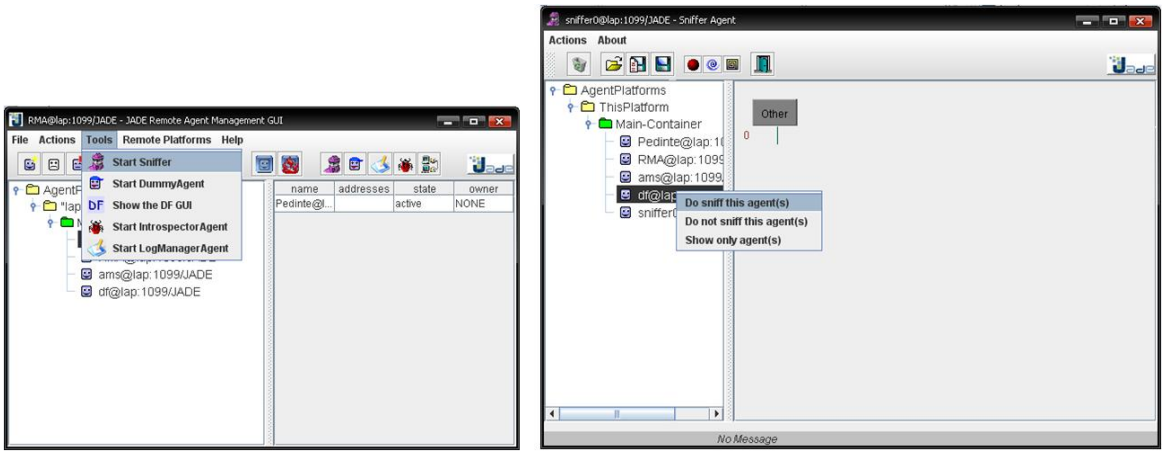

Figura 3.8: Execução do Agente Sniffer.

Vamos adicionar os agentes `Pedinte` e DF. Observe que a cada minuto o `Pedinte` envia uma mensagem para o DF buscando por agentes `Bombeiros`, e recebe uma mensagem de resposta que no caso está vazia, pois não temos agente `Bombeiro` na plataforma.

Vamos adicionar o agente `Bombeiro` à plataforma, com a seguinte linha:

```
java jade.Boot -container Bombeiro:Bombeiro
```

Adicionalmente, vamos adicionar este agente ao *Sniffer*. Agora, quando o agente `Pedinte` enviar uma mensagem para o DF, este responderá com uma mensagem contendo a AID do agente `Bombeiro` e então, o agente `Pedinte` envia uma mensagem diretamente para o agente `Bombeiro`. O resultado do contexto apresentado está ilustrado na Figura 3.9.

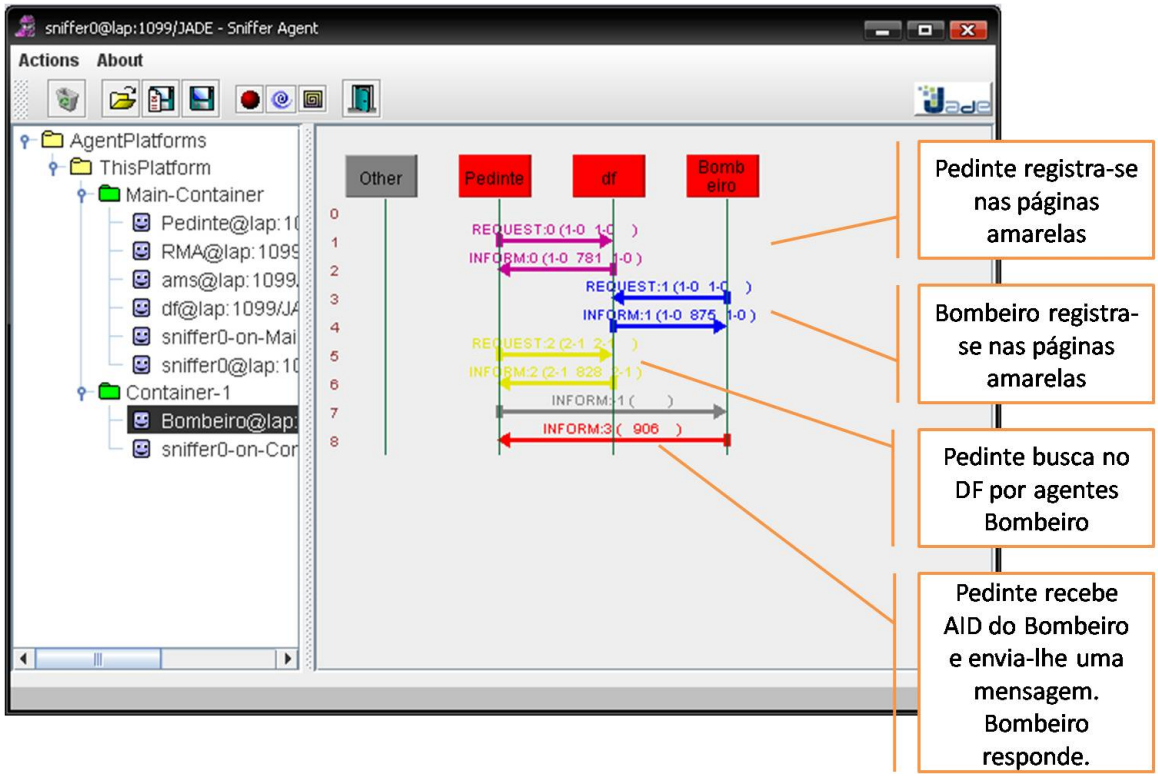

Figura 3.9: Troca de Mensagens entre os Agentes.

3.4.4 Notificação

Podemos também utilizar o serviço de notificação do DF para resolver este problema. Ao invés do agente `Pedinte` ficar realizando uma busca de novos agentes que oferecem determinado serviço a cada minuto, este pode pedir para que o DF notifique-o sempre quando um novo agente que oferece o serviço desejado se registrar nas páginas amarelas.

Para que isto seja possível, a plataforma JADE disponibiliza um serviço denominado *DF Subscription Service* que implementa um protocolo SUBSCRIBE entre os agentes envolvidos. O código desta funcionalidade está descrito na Caixa de Código 3.17. Todo o código fonte está comentado para que seja plausível o entendimento da implementação.

Código 3.17: Solicitante2.java

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.core.AID;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;
import jade.domain.FIPANames;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.proto.SubscriptionInitiator;

public class Solicitante2 extends Agent {

    protected void setup() {
        //Nada mudou neste método, apenas o método Busca()
        //passou a se chamado de PedeNotificação()

        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            String argumento = (String) args[0];
            if (argumento.equalsIgnoreCase("fogo")) {
                ServiceDescription servico = new ServiceDescription();
                servico.setType("apaga fogo");
                PedeNotificacao(servico, "fogo");
            }
            if (argumento.equalsIgnoreCase("ladrão")) {
                ServiceDescription servico = new ServiceDescription();
                servico.setType("prende ladrão");
                PedeNotificacao(servico, "ladrão");
            }
            if (argumento.equalsIgnoreCase("doente")) {
                ServiceDescription servico = new ServiceDescription();
                servico.setType("salva vidas");
                PedeNotificacao(servico, "doente");
            }

            //Comportamento para receber mensagens
            addBehaviour(new CyclicBehaviour(this) {
                /*As mensagens de notificação do DF atendem ao protocolo
                 * FIPA Subscribe, e elas possuem um método exclusivo para
                    sua recepção. (SubscriptionInitiator)
                 * Então, devemos ler todas as mensagens, exceto as que
                    obdecem o protocolo
                 * FIPA Subcribe.*/
                //Faço um filtro para receber mensagens do protocolo
                   Subscribe
```

```
                MessageTemplate filtro = MessageTemplate.MatchProtocol(
                    FIPANames.InteractionProtocol.FIPA_SUBSCRIBE);
45              //Crio um novo filtro que realiza uma inversão lógica no
                    filtro anterior.
                //ou seja, não aceita mensagens do protocolo Subscribe
                MessageTemplate filtro2 = MessageTemplate.not(filtro);
48
                public void action() {
                    //Só recebe mensagens do filtro 2
51                  ACLMessage msg = receive(filtro2);
                    if (msg != null) {
                        System.out.println(msg.getSender() + " : " + msg.
                            getContent());
54                  }else
                    block();
                }
57          });
        }
    }
60  protected void PedeNotificacao(final ServiceDescription sd, final
        String Pedido) {

        //Crio descrição da entrada no registro
63      DFAgentDescription dfd = new DFAgentDescription();
        dfd.addServices(sd);

66      //Crio mensagem de notificação

        ACLMessage mgs = DFService.createSubscriptionMessage(this,
            getDefaultDF(), dfd, null);
69
        //Agora iniciamos o comportamento que ficará esperando pela
            notificação do DF

72          addBehaviour(new SubscriptionInitiator(this, mgs) {
            //A mensagem de notificação é uma mensagem INFORM, então
            //utilizo o método padrão handleInform(). Este é um método pré-
                definido para
75          //manipular mensagens do tipo INFORM.
            protected void handleInform(ACLMessage inform) {
                try {
78                  //Retorna array de AIDs dos Agentes
                    DFAgentDescription[] dfds = DFService.
                        decodeNotification(inform.getContent());
                    //Crio mensagem
81                  ACLMessage mensagem =new ACLMessage(ACLMessage.INFORM);
                    //Capturo AID do agente
```

```
                        mensagem.addReceiver(dfds[0].getName());
84                      //Defino conteúdo da mensagem
                        mensagem.setContent(Pedido);
                        //Envio a mensagem
87                      myAgent.send(mensagem);

                    } catch (FIPAException e) {
90                      e.printStackTrace();
                    }
                }
93          });
    }
}
```

Vamos executar o agente `Pedinte`, solicitando ajuda para um doente, e posteriormente um agente `Médico`. A troca de mensagens entre esses agentes e o DF está ilustrada na Figura 3.10.

```
java jade.Boot Pedinte:Solicitante2(doente)
java jade.Boot Medico:Medico
```

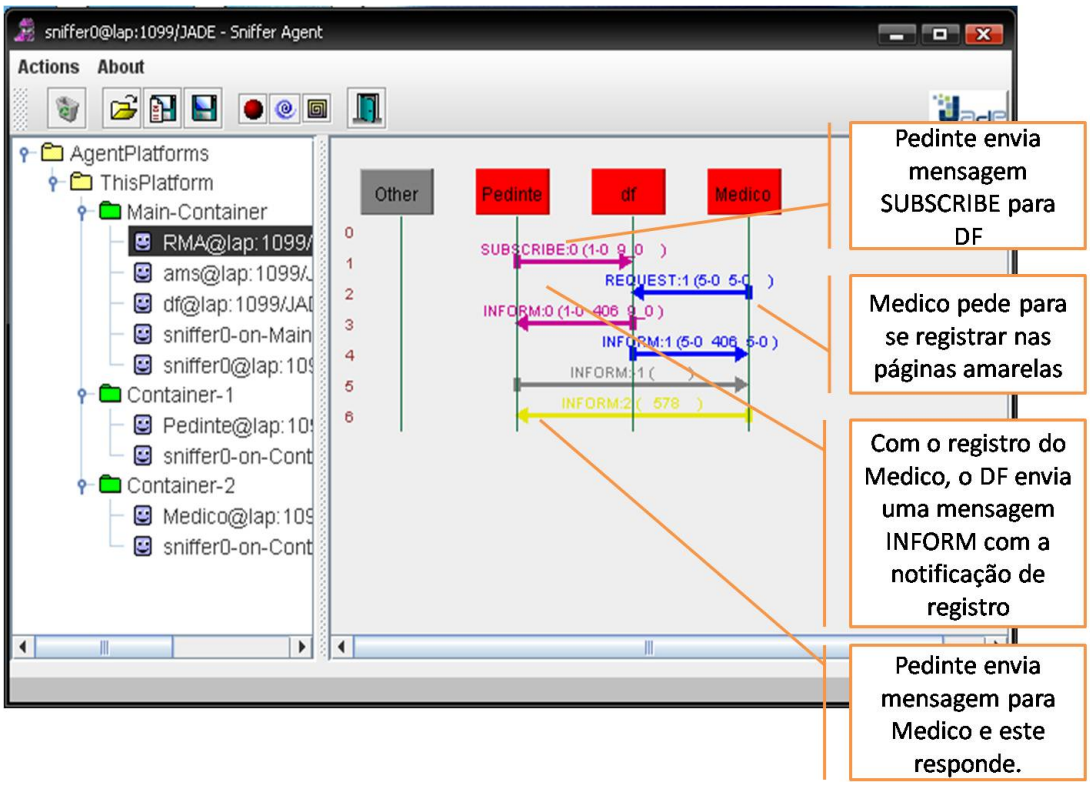

Figura 3.10: Páginas Amarelas - Notificação.

Observe que o pedido de notificação ao DF ocorre com a seguinte linha de código:

```
ACLMessage mgs = DFService.createSubscriptionMessage(this, getDefaultDF(),
dfd, null);
```

Da mesma maneira que pedimos a notificação, podemos pedir para que não sejamos mais notificados quando um determinado agente se registrar. Neste caso basta executar o método `createCancelMessage(Agent a, AID dfName, ACLMessage subscribe)`, onde os parâmetros são respectivamente o agente notificado, a AID do DF, e a mensagem de notificação inicial. No nosso exemplo, o objeto `msg` da classe `ACLMessage` será a mensagem de notificação e podemos pedir que não sejamos notificados com a seguinte linha (considerando que estamos dentro de um comportamento):

```
DFService.createCancelMessage(myAgent, getDefaultDF(), msg);
```

3.5 Páginas Brancas

Podemos consultar os agentes existentes na plataforma com uma busca nas páginas brancas da plataforma. De acordo com o padrão FIPA, quem realiza este serviço é o AMS (*Agent Management Service*). Para realizarmos a busca temos que importar as bibliotecas `jade.domain.AMSService`, `jade.domain.FIPAAgentManagement` e `jade.domain.AMSService`, para que os métodos de interação com o AMS estejam disponíveis. Uma busca nas páginas brancas dá-se pelo método `search()` da classe `AMSService`. Este método retorna um vetor de objetos `AMSAgentDescription`.

Por padrão o resultado desta busca retorna apenas um agente na plataforma. Para que possamos informar que desejamos obter todos os agentes da plataforma, devemos passar como parâmetro do método `search()` um objeto da classe `SearchConstraints`. Na Caixa de Código 3.18 temos um agente que busca por todos os agentes contidos na plataforma. A explicação da implementação está contida nos comentários do código.

Código 3.18: AgenteBuscaAMS.java

```
import jade.core.Agent;
import jade.core.AID;
import jade.domain.AMSService;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.FIPAException;


public class AgenteBuscaAMS extends Agent{

    protected void setup() {
        try {
            //Quero buscar quais agentes estão na plataforma
            AMSAgentDescription[] agentes = null;
            //Crio objeto SearchConstraints para definir
            //que desejo todos os resultados
            SearchConstraints c = new SearchConstraints();
            //O método setMaxResults indica o número de resultados
            //que desejo obter. Por definição, -1 significa todos.
            c.setMaxResults (new Long(-1));
            //busco pelos agentes
            //AMSService.search(agente que busca, vetor de retorno,
                características)
            agentes= AMSService.search(this, new AMSAgentDescription(), c);
            //Capturo minha AID
            AID myAID = getAID();
            for(int i=0; i<agentes.length; i++) {
                AID agenteID = agentes[i].getName();
                //Imprimo todos os agentes
                //Este agente será identificado com **** para diferenciar
```

```
                //dos demais
30              System.out.println((agenteID.equals(myAID)? "***": "    " )
                        + i + ": " + agenteID.getName() );
            }
33          } catch (FIPAException ex) {
                ex.printStackTrace();
            }
36          //Finalizo agente
            doDelete();
            //Finalizo aplicação
39          System.exit(0);

        }
42
}
```

Vamos executar a seguinte linha de comando:

```
java jade.Boot -gui
```

Com esta linha, conforme já vimos, executamos as ferramentas gráficas da plataforma. Mas também temos três agentes que são criados com o parâmetro `-gui`: o AMS, o DF e o RMA. Vamos agora executar o agente `Buscador`, que irá buscar nas páginas brancas quais agentes que estão na plataforma e imprimir a lista de agentes encontrados. Executamos o agente `Buscador` com a seguinte linha de comando:

```
java jade.Boot -container Buscador:AgenteBuscaAMS
```

E obtemos o seguinte resultado:

```
***0: Buscador@lap:1099/JADE
    1: RMA@lap:1099/JADE
    2: df@lap:1099/JADE
    3: ams@lap:1099/JADE
```

3.6 Protocolos de Interação

O padrão FIPA especifica um conjunto de protocolos que podem ser empregados na padronização das conversas entre os agentes. Para cada conversa, a plataforma JADE distingue entre dois papéis: o papel do iniciador, atribuído ao agente que inicia a conversa, e o papel do participante, representando o agente que responde ao ato comunicativo executado pelo iniciador.

JADE proporciona as classes `AchieveREIniciator` e `AchieveREResponder` para a implementação dos protocolos no estilo do FIPA-REQUEST, tais como:

- FIPA QUERY;
- FIPA REQUEST-WHEN;
- FIPA SUBSCRIBE.

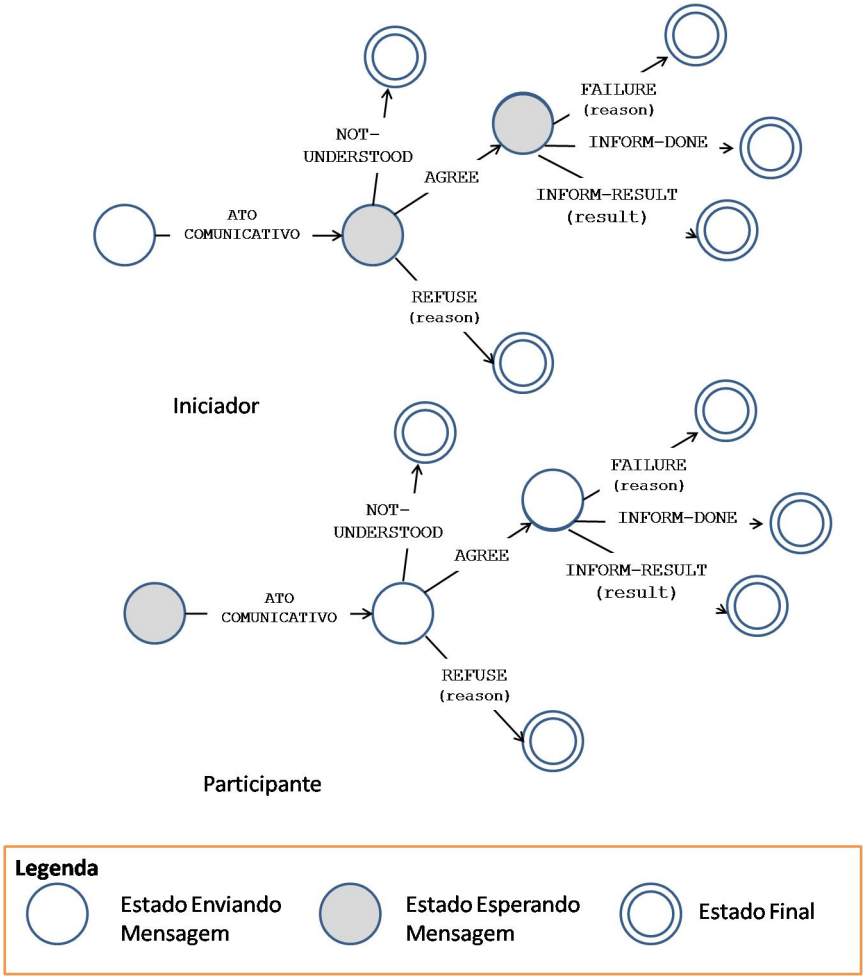

Figura 3.11: Estrutura dos Protocolos Baseados no FIPA-REQUEST.

O comportamento dessas classes é baseado em uma máquina de estados finito, semelhante ao *FSMBehaviour*. A Figura 3.11 ilustra a estrutura dos protocolos de interação do tipo FIPA-REQUEST. O iniciador envia uma mensagem com um determinado ato comunicativo. O participante pode responder um NOT-UNDERSTOOD ou um REFUSE, assim como pode enviar uma mensagem AGREE indicando que está disposto a realizar a ação do ato comunicativo.

O participante executa a ação e, finalmente, deve responder com uma mensagem INFORM indicando o resultado da ação, ou com uma mensagem FAILURE caso algo de errado ocorra.

3.6.1 Classe AchieveREInitiator

O iniciador envia uma mensagem aos agentes participantes e espera por suas respostas. Ainda que esteja esperando uma mensagem INFORM como resposta, deve estar também preparado para um REFUSE, NOT_UNDERSTOOD ou FAILURE.

Por exemplo, considere um agente que inicia um protocolo FIPA-REQUEST. Este deve construir uma mensagem ACL indicando a performativa (*request*) e o protocolo empregado. Com isto, basta informar o receptor o conteúdo da mensagem. Finalmente, o agente adiciona um comportamento do tipo `AchieveREInitiator` passando como referência o agente iniciador (ele, no caso) e a mensagem a ser enviada. Neste caso, nosso agente está preparado para receber mensagens INFORM deste protocolo, por isto o mesmo possui implementado o método `handleInform()`. Esta implementação está na Caixa de Código 3.19.

Código 3.19: AchieveREInitiator

```
ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setProtocol(FIPAProtocolNames.FIPA_REQUEST);
request.addReceiver(new AID("receptor", AID.ISLOCALNAME);
myAgent.addBehaviour( new AchieveREInitiator(myAgent, request) {
    protected void handleInform(ACLMessage inform) {
        System.out.println("Protocolo Finalizado.");
    }
});
```

3.6.2 Classe AchieveREResponder

Esta classe proporciona a implementação do papel do participante. É muito importante passar as configurações da mensagem como argumento no construtor desta classe, pois assim será possível saber quais tipos de mensagens ACL serão recebidas. Utiliza-se um *MessageTemplate* para criar o padrão de mensagens a ser recebida, informando o protocolo em que esta mensagem deve estar envolvida para poder ser lida.

A utilização da classe `AchieveREResponder` será implementada a seguir com a utilização de um exemplo.

3.6.3 Exemplo de Implementação do Protocolo FIPA-Request

O protocolo REQUEST é utilizado quando um agente solicita a outro que execute alguma ação. O participante pode aceitar ou recusar o pedido, e em caso de aceitá-lo, deverá

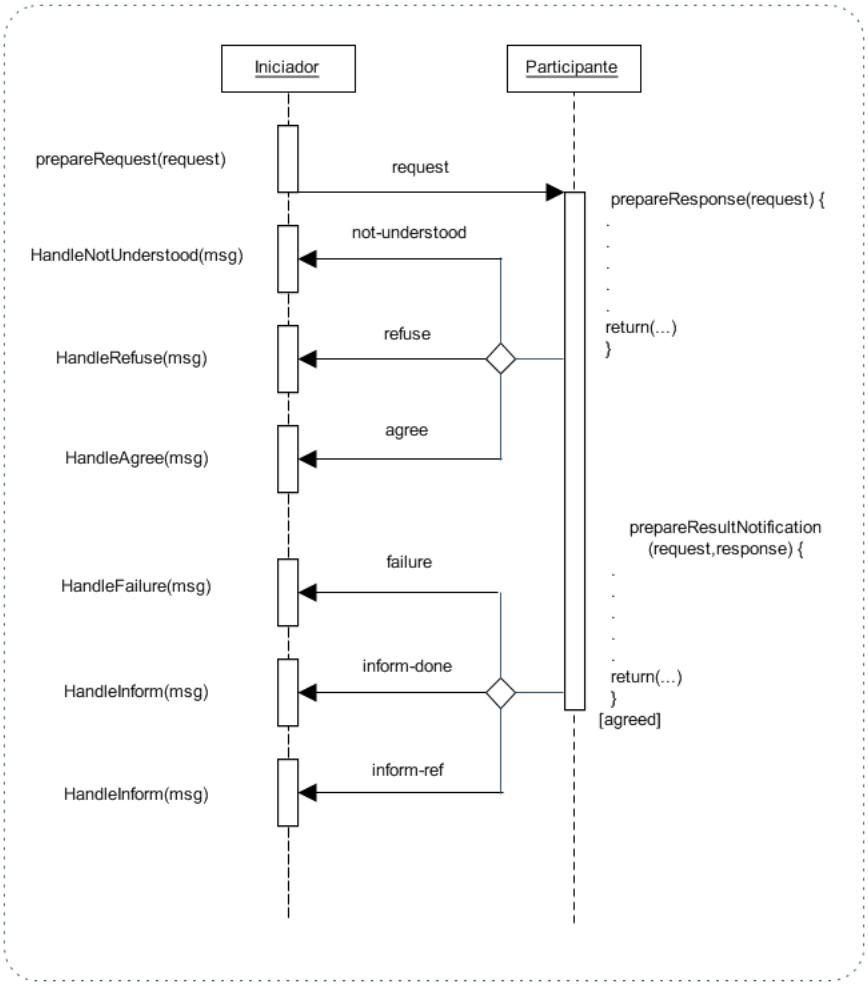

Figura 3.12: Protocolo FIPA-REQUEST.

realizar o que lhe foi pedido e indicar ao iniciador quando a execução do pedido estiver concluída.

A Figura 3.12 ilustra a troca de mensagens entre os agentes iniciador e participante no protocolo REQUEST. Observe que existe um método que representa uma ou mais respostas neste protocolo.

Vamos desenvolver uma aplicação mais robusta do nosso exemplo do agente `Alarmado` e do agente `Bombeiro`. Nesta aplicação não existe mais um agente `Bombeiro` e sim um agente `Central de Bombeiros`. A comunicação entre estes agentes será regida pelo protocolo REQUEST. O agente `Alarmado` (instância da classe `FIPARequestAlarmado`) avisa da existência de um incêndio a uma determinada distância aos agentes `Bombeiro`, que agora estão na `Central de Bombeiros` instância da classe `FIPARequestCentraldeBombeiros`. As centrais estão sempre alertas sobre chamadas informando sobre incêndios. Recebido

o aviso, cada central possui uma certa distância máxima que pode atuar e se a distância estiver dentro do limite permitido, a central irá apagar o fogo. Existe também uma probabilidade de 20% de faltar água para o combate ao incêndio.

O código das classes `FIPARequestAlarmado` e `FIPARequestCentraldeBombeiros` estão nas Caixas de Código 3.20 e 3.21, respectivamente. Para auxiliar o entendimento dos métodos, consulte a Figura 3.12 para entender o que cada método representa dentro do protocolo e seu momento de execução.

Código 3.20: FIPARequestAlarmado.java

```
import jade.core.Agent;
import jade.core.AID;
import jade.lang.acl.ACLMessage;
import jade.proto.AchieveREInitiator;
//para implementar o protocolo request importamos a seguinte classe:
import jade.domain.FIPANames;

public class FIPARequestAlarmado extends Agent {

    protected void setup() {

        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            System.out.println("Solicitando ajuda a várias centrais de
                bombeiros...");
            \\montando a mensagem a ser enviada posteriormente
            ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
            for (int i = 0; i < args.length; i++) {
               msg.addReceiver(new AID((String) args[i], AID.ISLOCALNAME));
            }
            msg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
            msg.setContent("Fogo a 5 kms");
            /* A classe Iniciador(abaixo) extende a classe
                AchieveREInitiator, ela
            atua como o iniciador do protoloco. Seu método construtor envia
            automaticamente a mensagem que está no objeto msg */
            addBehaviour(new Iniciador(this, msg));
        } else {
            System.out.println("Especifique o nome de pelo menos uma
                central de bombeiros");
        }

    }

    class Iniciador extends AchieveREInitiator {

```

```
//envia a mensagem request para os receptores que foram especificados no
    objeto msg
        public Iniciador(Agent a, ACLMessage msg) {
36          super(a, msg);  //parâmetros = agente que está enviando,
                mensagem a ser enviada
        }
//Os métodos a seguir tratam a resposta do participante
39      //Se o participante concordar, isto é, enviar uma mensagem AGREE
        protected void handleAgree(ACLMessage agree) {
            System.out.println("Central de bombeiros " + agree.getSender().
                getName() + " informa que saiu para apagar o fogo");
42      }

        //Se o participante se negar, enviando uma mensagem REFUSE
45      protected void handleRefuse(ACLMessage refuse) {
            System.out.println("Central de bombeiros " + refuse.getSender()
                .getName() + " responde que o fogo está muito longe " +
                    "e não pode apagá-lo");
48      }

        //Se o participante não entendeu, enviando uma mensagem NOT-
            UNDERSTOOD
51      protected void handleNotUnderstood(ACLMessage notUnderstood) {
            System.out.println("Central de bombeiros " + notUnderstood.
                getSender().getName() + "por algum motivo não entendeu a
                solicitação");

54      }

        //Se houve uma falha na execução do pedido
57      protected void handleFailure(ACLMessage failure) {
            //Verifica inicialmente se foi um erro nas páginas brancas
            if (failure.getSender().equals(getAMS())) {
60              System.out.println("Alguma das centrais de bombeiro não
                    existe");
            }
 /* O conteúdo de uma mensagem envolvida neste protocolo é automaticamente
     colocado entre parênteses. Com o método substring() podemos ler apenas
     o que está dentro deles.*/
63
             else {
                System.out.println("Falha na central de bombeiros " +
                    failure.getSender().getName() +
66              ": " + failure.getContent().substring(1, failure.getContent
                    ().length() - 1));
            }
        }
```

```
69
        //Ao finalizar o protocolo, o participante envia uma mensagem
            inform
        protected void handleInform(ACLMessage inform) {
72          System.out.println("Central de bombeiros" + inform.getSender().
                getName() + " informa que apagou o fogo");
        }
    }
75  }
```

Código 3.21: FIPARequestCentraldeBombeiros.java

```
    import jade.core.Agent;
    import jade.core.AID;
3   import jade.lang.acl.ACLMessage;
    import jade.proto.AchieveREInitiator;
    import jade.domain.FIPANames;
6   import jade.domain.FIPAAgentManagement.NotUnderstoodException;
    import jade.domain.FIPAAgentManagement.RefuseException;
    import jade.domain.FIPAAgentManagement.FailureException;
9   import jade.lang.acl.MessageTemplate;
    import jade.proto.AchieveREResponder;
    import java.util.StringTokenizer;
12
    public class FIPARequestCentraldeBombeiros extends Agent {

15      public double DISTANCIA_MAX;

        protected void setup() {
18
            DISTANCIA_MAX = (Math.random() * 10);
            System.out.println("Central " + getLocalName() + ": Aguardando
                alarmes...");
21          //Meu agente conversa sob o protocolo FIPA REQUEST
            MessageTemplate protocolo = MessageTemplate.MatchProtocol(FIPANames
                .InteractionProtocol.FIPA_REQUEST);
            MessageTemplate performativa = MessageTemplate.MatchPerformative(
                ACLMessage.REQUEST);
24          MessageTemplate padrao = MessageTemplate.and(protocolo,
                performativa);

            addBehaviour(new Participante(this, padrao));
27      }

        class Participante extends AchieveREResponder {
30
            public Participante(Agent a, MessageTemplate mt) {
                //Define agente e protocolo de comunicação
```

```
33              super(a, mt);
            }

36          /* Método que aguarda uma mensagem REQUEST, definida com o uso do
                objeto mt, utilizando no construtor desta classe.
            O retorno deste método é uma mensagem que é enviada automaticamente
                para o iniciador. */
            protected ACLMessage prepareResponse(ACLMessage request) throws
                NotUnderstoodException, RefuseException {
39
                System.out.println("Central " + getLocalName() + ": Recebemos
                    uma chamada de " + request.getSender().getName() +
                        " dizendo que observou um incêndio");
42

    /*A classe StringTokenizer permite que você separe
45  * ou encontre palavras (tokens) em qualquer formato.*/

                StringTokenizer st = new StringTokenizer(request.getContent());
48              String conteudo = st.nextToken(); //pego primeiro token
                if (conteudo.equalsIgnoreCase("fogo")) { //se for fogo
                    st.nextToken(); //pulo o segundo
51                  int distancia = Integer.parseInt(st.nextToken()); //capturo
                        DIST
                    if (distancia < DISTANCIA_MAX) {
                        System.out.println("Central " + getLocalName() + ":
                            Saimos correndo!");
54
                        ACLMessage agree = request.createReply();
                        agree.setPerformative(ACLMessage.AGREE);
57                      return agree;  //envia mensagem AGREEE
                    } else {
                        //Fogo está longe. Envia Mensagem Refuse com o motivo
60                      System.out.println("Central " + getLocalName() + ":
                            Fogo está longe demais. Não podemos atender a
                            solicitação.");
                        throw new RefuseException("Fogo está muito longe");
                    }
63              } //envia mensagem NOT_UNDERSTOOD
                else {
                    throw new NotUnderstoodException("Central de Bombeiros não
                        entendeu sua mensagem");
66              }

            }
69
            //Prepara resultado final, caso tenha aceitado
```

```
72          protected ACLMessage prepareResultNotification(ACLMessage request,
                ACLMessage response) throws FailureException {
                if (Math.random() > 0.2) {
                    System.out.println("Central " + getLocalName() + ":
                        Voltamos de apagar o fogo.");
75                  ACLMessage inform = request.createReply();
                    inform.setPerformative(ACLMessage.INFORM);
                    return inform; //envia mensagem INFORM
78              } else {
                    System.out.println("Central " + getLocalName() + ": Ficamos
                         sem água");
                    throw new FailureException("Ficamos sem água");
81              }

            }
84          }
    }
```

Na classe `FIPARequestCentraldeBombeiros` existe os seguintes códigos em seu método `setup()`:

```
MessageTemplate protocolo = MessageTemplate.MatchProtocol
(FIPANames.InteractionProtocol.FIPA_REQUEST);
MessageTemplate performativa=MessageTemplate.MatchPerformative
(ACLMessage.REQUEST);
MessageTemplate padrao=MessageTemplate.and(protocolo, performativa);
```

Um agente da classe `FIPARequestCentraldeBombeiros` implementa o papel de participante do protocolo REQUEST e, por isto, o mesmo implementa a classe `AchieveREResponder`, e no construtor desta classe faz-se necessário passar qual é o padrão de mensagens que ele está aguardando. Estas linhas de código criam um objeto `padrao` da classe `MessageTemplate` que irá fazer com que o agente aceite apenas mensagens do protocolo REQUEST e mensagens do tipo REQUEST.

Para execução deste cenário vamos utilizar um ambiente distribuído. Conforme já vimos, a plataforma JADE trabalha com o conceito de *containers*, representando um ambiente onde os agentes podem executar seu ciclo de vida. Onde os agentes estão rodando, deve possuir uma JRE e as bibliotecas da plataforma JADE para que estes funcionem perfeitamente. A Figura 3.13 ilustra este conceito.

Em uma plataforma sempre temos o *main-container*, aquele iniciado primeiro, composto pelo DF e AMS, além de outros agentes. Observe que quando iniciávamos outro agente em um mesmo computador, incluíamos o parâmetro `-container`, isto representa a criação de outro container.

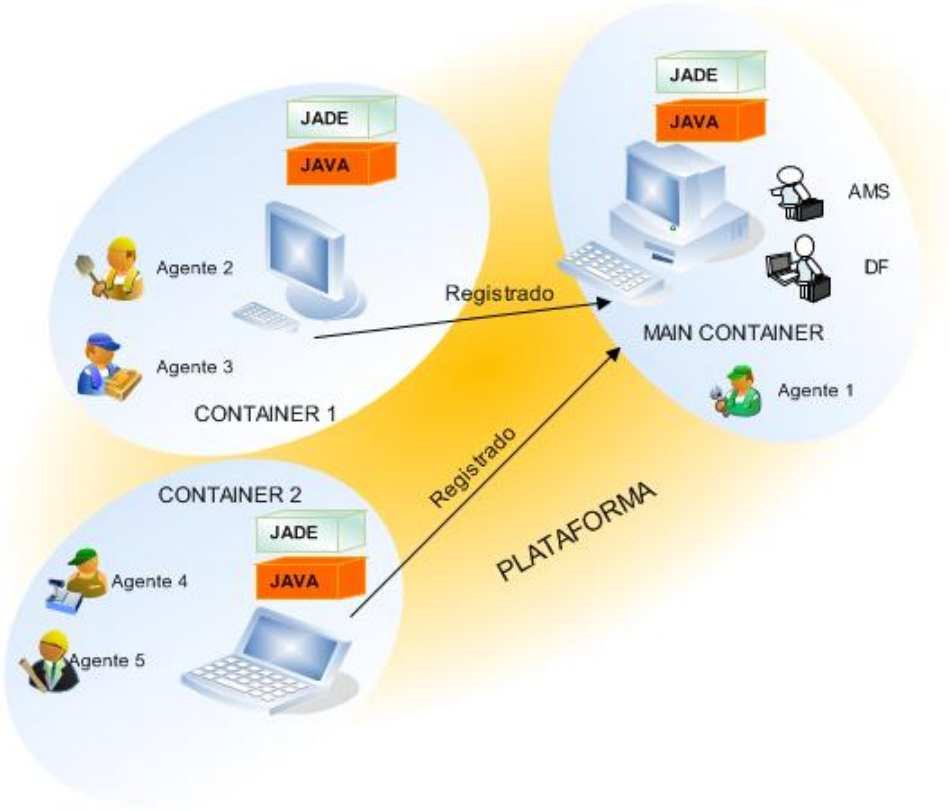

Figura 3.13: Plataforma Distribuída JADE.

O que vamos fazer nesta execução é alocar alguns agentes em uma máquina e outros em outra que estão ligadas em rede. O nosso agente `Alarmado` executará em um *container* em uma máquina da plataforma. As centrais dos bombeiros estarão na outra máquina da rede.

Vamos definir que a máquina que contará o *main-container* da plataforma é a máquina 1, que contém as centrais. Vamos iniciar as centrais no *main-container*. Para isto, em nossa máquina 1 (denominada de `PC-1`) executamos a seguinte linha:

```
java jade.Boot C1:FIPARequestCentraldeBombeiros
               C2:FIPARequestCentraldeBombeiros
               C3:FIPARequestCentraldeBombeiros
```

Com isto, temos o seguinte resultado:

```
Central C1: Aguardando Alarmes...
```

```
Central C2: Aguardando Alarmes...
Central C3: Aguardando Alarmes...
```

Na outra máquina da rede (PC-2) executamos o agente Alarmado, com a seguinte linha de comando:

```
java jade.Boot -host PC-1 -container Alarmado:FIPARequestAlarmado(C1 C2 C3)
```

Com esta linha de comando, informamos onde o *main-container* está, no caso o parâmetro -host indica que este está no computador PC-1. Com a execução do agente Alarmado, observamos no *prompt* da máquina PC-1:

```
Central C1: Recebemos uma chamada de Alarmado@PC-1:1099/JADE dizendo
 que observou um incêndio.
Central C2: Recebemos uma chamada de Alarmado@PC-1:1099/JADE dizendo
que observou um incêndio.
Central C3: Recebemos uma chamada de Alarmado@PC-1:1099/JADE dizendo
 que observou um incêndio.
```

Observe que mesmo estando na máquina PC-2, as mensagens indicam que o agente Alarmado está no PC-1. Isto ocorre pois o nome padrão de uma plataforma é o nome da máquina onde está o *main-container*. Mas o agente Alarmado está executando na máquina PC-2.

Agora temos o seguinte resultado, de acordo com as probabilidades definidas para nossos agentes centrais:

```
Central C1: Saimos correndo!
Central C2: Fogo está longe demais. Não podemos atender a solicitação.
Central C3: Fogo está longe demais. Não podemos atender a solicitação.
```

Neste mesmo instante, no *prompt* do PC-2 observamos o seguinte:

```
Central de bombeiros C1@PC-1:1099/JADE informa que saiu para apagar o fogo.
Central de bombeiros C2@PC-1:1099/JADE informa que o
 fogo está muito longe e não pode apagá-lo.
Central de bombeiros C3@PC-1:1099/JADE informa que o
fogo está muito longe e não pode apagá-lo.
```

Agora, como a Central de Incêndio C1 aceitou apagar o fogo, ela notifica que o fogo está apagado para o agente alarmado. Obtemos a seguinte linha:

```
Central de bombeiros C1@PC-1:1099/JADE informa que apagou o fogo.
```

Referências Bibliográficas

[Bellifemine, Caire e Greenwood 2007]BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. [S.l.]: John Wiley & Sons, 2007. ISBN 0470057475.

[Blaya 2005]BLAYA, J. A. B. *Tutorial básico de JADE*. [S.l.], Febrero 2005.

[JADE Wikispaces]JADE WIKISPACES. *Programacíon JADE*. [S.l.]. Disponível em: <http://programacionjade.wikispaces.com>. Acesso em: 22 Dez 2007.

[Vaucher e Ncho]VAUCHER, J.; NCHO, A. *JADE Tutorial and Primer*. [S.l.]. Disponível em: <www.iro.umontreal.ca/ vaucher/Agents/Jade/JadePrimer.html>. Acesso em: 22 Dez 2007.