# 1. Instalación

La versión 3.2 de JADE (*Java Agents Development Environment*) puede bajarse de la siguiente dirección

`http://jade.tilab.com`

aunque para hacerlo hay que rellenar un formulario. Si quereis instalarla en casa debereis hacerlo así. Aquí la vamos a obtener desde el servidor de ficheros del centro de cálculo de la facultad. La distribución está dividida en varios ficheros `zip` que son los siguientes:

- JADE-all-3.2.zip: un zip que, a su vez, alberga los siguientes cuatro fichero que aparecen a continuación.
- JADE-doc-3.2.zip: la documentación javadoc, el manual del administrador, el del programador y un tutorial.
- JADE-src-3.2.zip: el código fuente sin compilar.
- JADE-bin-3.2.zip: el código ya compilado y listo para ser interpretado.
- JADE-examples-3.2.zip: ejemplos de uso de la plataforma.

Lo único que necesita JADE para funcionar es una versión correcta del Java Run Time Environment, la 1.4. Una vez hemos obtenido en nuestra cuenta el fichero `JADE-all-3.2.zip`, creamos un directorio a parte para jade, colocamos allí la distribución y hacemos:

```
>unzip JADE-all-3.2.zip
>unzip JADE-doc-3.2.zip
>unzip JADE-src-3.2.zip
>unzip JADE-bin-3.2.zip
>unzip JADE-examples-3.2.zip
```

y ya está todo el software desplegado.

Como podrá comprobarse se ha creado un directorio `jade` y bajo este directorio un directorio `lib` que es en donde están los jars necesarios para ejecutar la plataforma.

# 2. Arquitectura de JADE

Como sabeis, JADE responde a los estándares FIPA de organización del sistemas multi-agente y de comunicación entre agentes. Esto quiere decir que las especificaciones que incluye la arquitectura abstracta se van a encontrar también aquí.

Según FIPA, los agentes son entidades software que están localizadas en una única plataforma. Una plataforma de agentes es un entorno de ejecución en

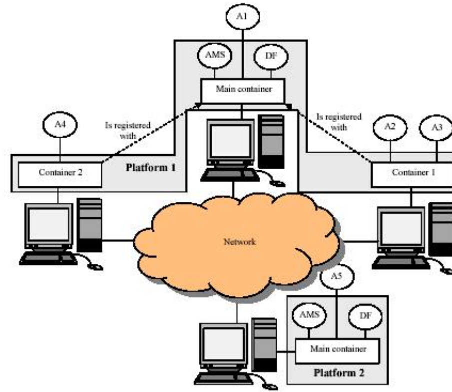


Figura 1: Ejemplo en el que aparecen varios contenedores, distribuidos en sus correspondientes plataformas

donde los agentes están ejecutándose y a través del cual disponen de un servicio de páginas blancas para buscar otros agentes, un servicio de páginas amarillas para buscar servicios que otros agentes ofrecen, un módulo de gestión a través del cual se accede a estas facilidades y por último un sistema de envío/recepción de mensajes mediante el cual el AMS puede hacer llegar a su destino los mensajes generados por los agentes ubicados en la misma plataforma.

A parte de todos estos detalles propios de FIPA, JADE incluye la noción de contenedor. Un contenedor es una instancia del entorno de ejecución de JADE. En un contenedor es posible albergar un número indeterminado de agentes. Existe un tipo especial de contenedor denominado *principal* (i.e. main container). Debe existir uno y solo uno de estos contenedores por cada plataforma FIPA de agentes y el resto de contenedores de la plataforma, una vez ejecutados (i.e. una vez ejecutado el correspondiente entorno de ejecución JADE) deben subscribirse al principal, por lo que el responsable de ejecutarlos también es responsable de indicar en dónde se localiza el contenedor principal. Vamos a aclarar todos estos conceptos con la figura 1, extraida del manual del administrador de JADE Véase como en la plataforma número 1 existen tres contenedores. En el principal están los elementos imprescindibles de cualquier plataforma FIPA, el AMS y el DF (este último opcional en la recomendación correspondiente aunque en JADE está siempre disponible). También existe un agente, el A1. Dentro de esta misma plataforma hay otros dos containers normales, el de la izquierda albergando al agente A4 y el de la derecha a los agentes A2 y A3. La segunda plataforma está únicamente formada por un contenedor principal y no más, albergando también el correspondiente sistema gestor de agentes y el DF, junto con el agente A5. Todos los agentes de la figura pueden comunicarse, siempre que cada agente que quiere hacerlo sepa el nombre del agente con quien lo quiere hacer. Por ejemplo, los agentes A2 y A3, en el mismo contenedor pueden comu-

nicarse. También pueden hacerlo los agentes A1 y A2, en distintos contenedores de la misma plataforma. Por último, también pueden A4 y A5, en distintas plataformas.

## 3. El ejemplo del tutorial

En el tutorial "JADE Programming for Beginners" aparece un ejemplo de un entorno de compra-venta de libros. Vamos a seguir en la primera práctica este tutorial. En ese entorno, existen unos agentes que venden libros y otros que los compran, en representación de sus correspondientes usuarios.

Los agentes compradores reciben órdenes de los usuarios a través de la línea de comandos. De tal forma que cuando un usuario quiere comprar un libro, introduce su título de esa manera. Tras esto, el agente requiere ofertas del libro de todos los agentes vendedores que conoce, periódicamente. El algoritmo es bastante simple pues se compra el libro al agente que manda la primera oferta. Si más de un agente manda una oferta, el comprador elige aquella con el precio más barato. Una vez comprado el libro, el agente comprador finaliza su ejecución.

Cada agente vendedor va a tener una interface mínima, a través de la cual se le pueden indicar los títulos que puede vender y sus correspondientes precios. Este tipo de agente espera indefinidamente peticiones de compra de los correspondientes agentes compradores. Cuando se le pide una oferta sobre un libro, lo busca en su catálogo local y si lo tiene, responde con el precio. Si no, rechaza la petición. Una vez se ha vendido un libro, lo elimina de su catálogo.

Obsérvese que en la distribución de JADE que acabamos de instalar, existía un fichero zip con el nombre JADE-examples-3.2.zip. Al desplegarlo podemos encontrar este ejemplo en el paquete de código Java

`examples.bookTrading`

## 4. Creando los agentes JADE

Para trabajar con agentes se debe conocer más o menos a fondo la clase

`jade.core.Agent`

ya que para crear un agente JADE se debe crear una nueva clase de agente que herede de ésta y que como mínimo implemente el métod `setup()`. Véase como se crea un agente comprador, mínimamente, con el siguiente código:

```
import jade.core.Agent;

public class BookBuyerAgent extends Agent {

  protected void setup() { // Printout a welcome message
    System.out.println(``Hallo! Buyer-agent `` + getAID().getName()
    + `` is ready.'');
```

```
  }
}
```

El método `setup()` se ejecuta cuando se va a lanzar el agente para su ejecución y únicamente debe contener código relativo a tareas de inicialización. Por lo tanto, de momento, el agente no va a hacer absolutamente nada más que emitir ese mensaje cuando se lance.

## 4.1. Identificadores de agentes

Como ya sabemos de la norma FIPA, cada agente debe tener un nombre, que forma parte de la descripción del agente que, a su vez, contiene más datos relativos a cómo mandarse mensajes.

En JADE, la descripción del agente, el AID (*Agent Identifier*) tiene su correspondiente clase java, la clase

`jade.core.AID`

a partir de la cual se pueden manejar los nombres y las demás informaciones con facilidad. La clase `Agent` incorpora el método `getAID()` que permite recuperar el nombre del agente. El nombre del agente, un identificador único globalmente, va a tener la estructura

```
<nickname>@nombre-plataforma>
```

Por lo tanto, el agente A1, localizado en la plataforma P1 va a tener el nombre `A1@P1`. Hablando de las direcciones que forman parte del AID, solamente tendremos que preocuparnos de ellas cuando el agente con el que queramos contactar esté en otra plataforma. Una vez que conocemos el nombre de pila de un agente, su AID puede obtenerse de la siguiente manera:

```
String nickname =  A1;
AID id = new AID(nickname, AID.ISLOCALNAME);
```

en donde con el argumento `AID.ISLOCALNAME` indicamos que el primer argumento es el nombre de pila y no el nombre único globalmente.

## 4.2. Ejecución de agentes en JADE

Una vez hemos creado el agente, ya estamos en condiciones de compilarlo. Ejecutamos:

```
javac -classpath <jars de jade> BookBuyerAgent.java
```

Y ya está.

Ahora ya podemos ejecutarlo. Para eso, tenemos que lanzar la plataforma pasándole como argumentos el nombre de pila del agente y, por supuesto, la clase Java en donde se implementa tal que así:

```
java -classpath <jars de jade> jade.Boot comprador:BookBuyerAgent
```

El resultado es el siguiente:

```
This is JADE snapshot - 2003/10/24
13:43:39 downloaded in Open Source, under LGPL restrictions, at
http://jade.cselt.it/
IOR:000000000000001149444C3A464950412F4D54533A312E30000000
0000000001000000000000600000010200000000A3132372E302E302E
310009A600000019AFABCB0000000002BCE5528F000000080000000000
0000000A000000000000010000000100000020000000000001000100000
000205010001000100200001010900000001000010100
Agent container Main-Container@JADE-IMTP://maquina.dif.um.es is ready.
Hallo! Buyer-agent comprador@maquina.dif.um.es:1099/JADE is ready.
```

La primera parte del mensaje impreso en pantalla es la información que JADE siempre genera sobre la licencia del software y demás. Dado que el protocolo de comunicación por defecto entre plataformas es IIOP, como ya mencionamos en clase, lo que se genera a continuación es la dirección IIOP en forma de IOR del correspondiente objeto distribuido de la plataforma con el que nos podemos comunicar remotamente.

Lo que aparece seguidamente es la indicación de que el contenedor principal está listo. y que está corriendo en la máquina del departamento de informática de la UMU. El siguiente mensaje viene de la ejecución del método `setup()` del agente comprador lo que indica que está todo listo. Obsérvese que, aunque el agente no va a hacer nada más, permanece ejecutándose hasta que se invoca el método `doDelete()`. Este método llama a su vez a `takeDown()` que podeis redefinir para incluir código relativo a limpieza de elementos del agente.

## 4.3. Pasando valores a un agente

Dado que el agente comprador debe saber el título del libro que tiene que comprar, esto ha de indicársele de alguna forma. Vamos a ver cómo podemos hacerlo de manera fácil mediante el empleo de un array de objetos que lleva incorporado la clase `Agent`. Vamos a modificar el código de nuestro comprador como sigue:

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

public class BookBuyerAgent extends Agent {
  // The title of the book to buy
  private String targetBookTitle;
  // The list of known seller agents
  private AID[] sellerAgents = {
    new AID(``seller1'', AID.ISLOCALNAME),
    new AID(``seller2'', AID.ISLOCALNAME)};
```

```
  // Put agent initializations here
  protected void setup() {
    // Printout a welcome message
    System.out.println(‘‘Hallo! Buyer-agent ‘‘ + getAID().getName() +
    ‘‘ is ready.‘‘);

    // Get the title of the book to buy as a start-up argument
    Object[] args = getArguments();
    if (args != null && args.length > 0)
    {
      targetBookTitle = (String) args[0];
      System.out.println( Trying to buy  +targetBookTitle);
    } else
    {
      // Make the agent terminate immediately
      System.out.println( No book title specified );
      doDelete();
    }
  }

  // Put agent clean-up operations here protected void
  takeDown() {
    // Printout a dismissal message
    System.out.println(‘‘Buyer-agent ‘‘ + getAID().getName() + ‘‘
    terminating.’’);
  }
}
```

La parte que tiene que ver con la recogida de argumentos está en el método `setup()`. Véase que el método `getArguments()` que devuelve un `Object[]` es el que proporciona los argumentos. Para que no haya ningún problema, el lector de los argumentos debe saber el número y el tipo de cada uno de los argumentos.

Ahora ha llegado el momento de compilar y ejecutar:

```
C:\jade>java jade.Boot buyer:BookBuyerAgent(The-Lord-of-the-rings)
This is JADE snapshot - 2003/10/24 13:43:39
downloaded in Open Source, under LGPL restrictions, at
http://jade.cselt.it/
IOR:000000000000001149444C3A464950412F4D54533A312E30000000000000000
1000000000000 0060000102000000000A3132372E302E302E310009A600000019A
FABCB0000000002BCE5528F0000 0008000000000000000000A00000000000001000
000010000002000000000000100010000000020501 0001000100200001010900000
00100010100
Agent container Main-Container@JADE-IMTP://IBM8695 is ready.
Hallo! Buyer-agent buyer@IBM8695:1099/JADE is ready.
Trying to buy The-Lord-of-the-rings
```

## 5. Especificando tareas mediante los comportamientos JADE

Las tareas o servicios que un agente hace realmente se especifican en comportamientos. Un comportamiento (i.e. behaviour) hace referencia a una funcionalidad que incorpora el agente. El código JAVA que implementa esa funcionalidad se incluye en una nueva clase creada para tal efecto, que ha de heredar de la clase

`jade.core.behaviours.Behaviour`

Una vez que se ha implementado, es suficiente para que el agente ejecute ese comportamiento que se invoque, en el cuerpo de acción del agente el método `addBehaviour` perteneciente a la clase `Agent`. Toda clase que herede de la de comportamiento deberá implementar el méotod `action()` en donde se debe incluir el código de las acciones correspondientes. También el método `done()` se debe implementar. Devuelve un booleano y sirve al agente para determinar si el comportamiento ha finalizado o no. Deberá devolver `true` en ese caso.

Podemos pensar que los behaviours son como los hilos de ejecución JAVA. Igual que las threads en Java, en un agente pueden ejecutarse a la vez tantos comportamientos como sea necesario. Sin embargo, a diferencia de las threads en JAVA, el decidir qué comportamiento se ejecuta en cada momento es tarea nuestra (en JAVA lo decide la máquina virtual). Esto es así para que cada agente equivalga únicamente a un único thread, con el consiguiente ahorro de ciclos de CPU y memoria que esto implica. En la figura 5 puede verse el camino por el que atraviesa un agente desde que comienza su ejecución hasta que finaliza y se elimina. Como puede verse, lo primero a ejecutar es el método `setup()`. Tras esto se comprueba que el agente sigue vivo y después se selecciona el siguiente comportamiento a ejectuar del conjunto de comportamientos que aun le quedan por ejecutar al agente. Se ejecuta su método `b.action()` y tras esto se pregunta si ha finalizado. Es posible que no lo haya hecho ya que un comportamiento puede ser o un simple trozo de código que se ejecuta una sola vez o bien varias veces dependiendo de otros factores. Si está ejecutado se elimina del conjunto de comportamientos del agente y no vuelve a ejecutarse. En todo caso, se vuelve a comenzar. Con todo esto, téngase en cuenta que un comportamiento como el siguiente

```
public class OverbearingBehaviour extends Behaviour
{
  public void action() {
    while (true) {
    // do something
    }
  }
  public boolean done() {
    return true;
  }
```

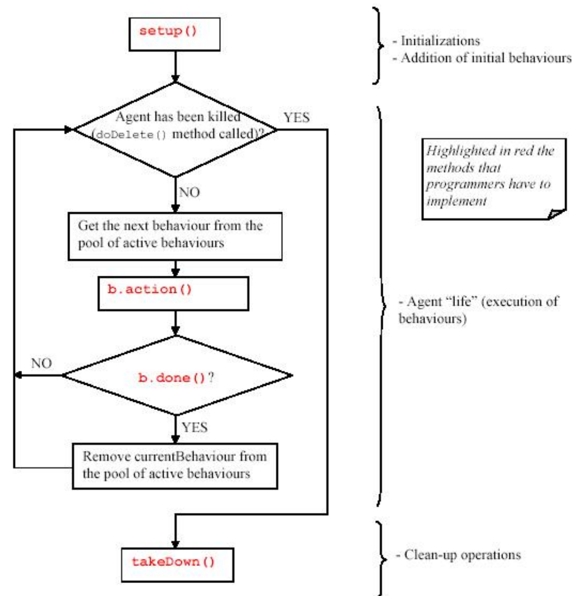


Figura 2: El ciclo de ejecución de un agente

```
}
```

causa que el agente no ejecute ningún comportamiento más durante toda su ejecución ya que el método `action()` no finaliza nunca.

## 5.1. Algunos comportamientos útiles

Podemos agrupar los behaviours JADE en tres grandes grupos:

1. Comportamientos *one-shot*: tipos de comportamiento los cuales se ejecutan de manera casi instantánea, y solamente una vez.

2. Comportamientos cíclicos: son aquellos que nunca son sacados del conjunto de comportamientos del agente y cuyo método `action()` siempre ejecuta el mismo código. Por lo tanto, nunca finalizan.

3. Comportamientos genéricos: algo más complejos, el código que se ejcuta en ellos depende del estatus del agente, y eventualmente finalizan su ejecución.

El siguiente es un ejemplo de comportamiento *one-shot*:

```
public class MyOneShotBehaviour extends OneShotBehaviour
```

```
{
  public void action() {
    // perform operation X
  }
}
```

y el siguiente un comportamiento de tipo cíclico: public class MyCyclicBehaviour extends CyclicBehaviour public void action() // perform operation Y en el que, en teoría, habría que implementar el método `done()` de tal forma que devolviera `false` pero ya lo implementa la clase `CyclicBehaviour`.

Por último, un ejemplo de comportamiento genérico:

```
public class MyThreeStepBehaviour extends Behaviour
{
  private int step = 0;

public void action() {
    switch (step) {
      case 0:
        // perform operation X
        step++;
    break;
      case 1:
        // perform operation Y
        step++;
    break;
      case 2:
        // perform operation Z
        step++;
    break;
    }
  }

  public boolean done() {
    return step == 3;
  }
}
```

Otro tipo de comportamientos JADE son los compuestos, que permiten combinar comportamientos previamente definidos de manera conveniente para aplicarles un orden de ejecución determinado como secuencial con `SequentialBehaviour`, en paralelo con `ParallelBehaviour` y guiados mediante un autómata finito determinista con `FSMBehaviour`. Véase el manual del programador de JADE para una descripción de estos comportamientos.

Otros tipos de comportamientos interesantes son `WakerBehaviour` y `TickerBehaviour`. Tienen en común que su ejecución es diferida. Se invocan y guardan hasta que se ha cumplido un determinado time-out. El primer solamente se ejecuta una vez y el segundo es un comportamiento cíclico.

El siguiente agente incorpora un comportamiento del primer tipo, el cual se ejecutará después de 10 segundos:

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

Obsérvese que ahora, en lugar de implementar el método `action()`, es el método `handleElapsedTimeOut()` el que albergará el código a ejecutar tras el cumplimiento del tiempo.

Un agente con el comportamiento `TickerBehaviour` se programará de manera similar:

```
public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new TickerBehaviour(this, 10000) {
      protected void onTick() {
        // perform operation Y
      }}
    );
  }
}
```

y el código a ejecutar cíclicamente se incluirá en el método `onTick()`.

## 5.2. Comportamientos del agente comprador

Recordemos que el agente comprador tiene un comportamiento cíclico, encargado de preguntar al agente vendedor si tiene el libro que desea comprar. Pues para esa parte de su comportamiento global añadiremos un `TickerBehaviour`, en el método `setup()` del agente comprador tal que así:

```
protected void setup() {
  // Printout a welcome message
  System.out.println(``Hallo! Buyer-agent ``  + getAID().getName() +
    `` is ready.'');
  // Get the title of the book to buy as a start-up argument
  Object[] args = getArguments();
  if (args != null && args.length > 0)
  {
    targetBookTitle = (String) args[0];
```

```
    System.out.println(``Trying to buy `` + targetBookTitle);
    // Add a TickerBehaviour that schedules a request to seller agents
    //every minute
    addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick() {
        myAgent.addBehaviour(new RequestPerformer());
      }}
    );
  } else
  {
    // Make the agent terminate
    System.out.println(``No target book title specified'');
    doDelete();
  }
}
```

Obsérvese que ahora estamos utilizando una variabe, `myAgent`, accesible dentro del comportamiento. Todos los comportamientos JADE incorporan esta variable para acceder a los datos del agente que está ejecutándolo en tiempo de ejecución. Este código se va a encargar de ejecutar cuando procede el método `RequestPerformer()` que es el que incorpora la lógica encargada de contactar con el vendedor e intentar comprar el libro que se pasa como argumento.

## 5.3. Comportamientos del agente vendedor

El agente vendedor es un poco más complicado que el comprador ya que, básicamente, debe realizar dos tareas. Una para responder a la pregunta de si se dispone o no del libro que el comprador quiere comprar. La otra encargada de ejectuar la venta del libro. Para cada una de esas tareas vamos a implementar sendos comportamientos cíclicos e independientes. El cómo están programados cada uno de los comportamientos lo veremos posteriormente.

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

public class BookSellerAgent extends Agent {
  // The catalogue of books for sale
  //(maps the title of a book to its price)
  private Hashtable catalogue;

  // The GUI by means of which the user can add books in
  //the catalogue
  private BookSellerGui myGui;

  // Put agent initializations here
  protected void setup() {
```

```
    // Create the catalogue
    catalogue = new Hashtable();

    // Create and show the GUI
    myGui = new BookSellerGui(this);
    myGui.show();

    // Add the behaviour serving requests for
    //offer from buyer agents
    addBehaviour(new OfferRequestsServer());

    // Add the behaviour serving purchase orders from
    //buyer agents
    addBehaviour(new PurchaseOrdersServer());
  }
  // Put agent clean-up operations here
  protected void takeDown() {
    // Close the GUI
    myGui.dispose();

    // Printout a dismissal message
    System.out.println(``Seller-agent `` + getAID().getName() +
      `` terminating.'');
  }
  /** This is invoked by the GUI when the user adds a new book for
    sale */
  public void updateCatalogue(final String title, final int price) {
    addBehaviour(new OneShotBehaviour() {
      public void action() {
        catalogue.put(title, new Integer(price));
      }
    }
    );
  }
}
```

Como podemos comprobar, el ejemplo se va complicando poco a poco. Obsérvese que se han añadido dos nuevos comportamientos que son

`OfferRequestsServer()` y el `PurchaseOrdersServer()`

que se encargan de servir las ofertas a petición de los compradores y de servir las compras, también a petición de los compradores, respectivamente. El método `updateCatalogue()` se encarga de añadir un comportamiento de tipo `one-shot` añadiendo un nuevo título al catálogo. Este método se invoca desde el interface, que se implementa fácilmente mediante una clase JADE, y se encarga añadir el nuevo título, introducido desde la correspondiente ventana.

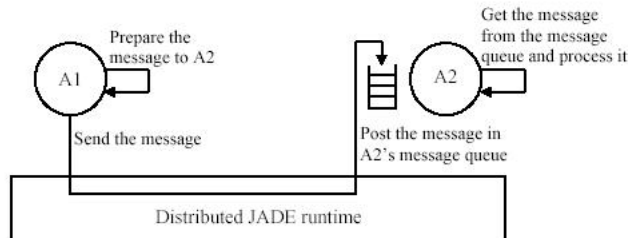


Figura 3: Esquema de transferencia asíncrona de mensajes en JADE

## 5.4. Cómo los agentes comprador y vendedor se comunican entre sí

JADE sigue el estándar FIPA. Como tal es capaz de hacer que mensajes del tipo ACL puedan intercambiarse entre los agentes implicados, a través del AMS (*Agent Management System*) y mediante un protocolo estándar como, por ejemplo, IIOP.

El esquema de envío de mensajes en JADE responde al de la figura 3. Como puede verse, en la figura aparece una cola de mensajes por cada agente. Cuand un mensaje llega al agente, solamente tiene que mirar en su cola de mensajes y extraerlo de ahí.

Para mandar mensajes en JADE, teniendo en cuenta cómo están definidos los mismos en FIPA, se debe hacer uso de la clase `ACLMessage` como en el ejemplo siguiente:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID(``Peter'', AID.ISLOCALNAME));
msg.setLanguage(``English'');
msg.setOntology(``Weather-forecast-ontology'');
msg.setContent(``Today it s raining'' );
send(msg);
```

en el que, como puede verse, el acto comunicativo es del tipo `inform`, se envía al agente `Peter`, en lenguaje de contenido del mensaje es inglés natural, la ontología respectiva es `Weather-forecast-ontology`, y el mensaje es el que aparece en la penúltima línea de código. Para mandar el mensaje hacemos uso del método `send()` de la clase `Agent`.

Si volvemos al ejemplo de compra-venta de libros, el agente comprador necesita un título para el usuario. Por lo tanto, debe buscarlo entre todos los posibles vendedores. Por ello, podemos hacer uno de un acto comunicativo del tipo `cfp` que se habrá de enviar a todos los agentes vendedores tal que así:

```
// Message carrying a request for offer ACLMessage
cfp = new ACLMessage(ACLMessage.CFP);
```

```
for (int i = 0; i < sellerAgents.lenght; ++i) {
  cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

Obsérvese que podemos añadir al mensaje tantos receptores como deseemos antes de enviarlo.

Ahora, para que el agente vendedor reciba las posibles ofertas, simplemente debe mirar en su cola de mensajes y coger el primero de ellos con el método `receive()`:

```
ACLMessage msg = receive();
if (msg != null) {
  // Process the message
}
```

teniendo en cuenta que el método mencionado no es bloqueante y, por lo tanto, si no se dispone de ningún mensaje en la cola, devolverá un valor nulo.

Siguiendo este sencillo ejemplo podemos programar el comportamiento correspondiente a `OfferRequestServer` que se encarga de responder a peticiones de libros de agentes compradores con el correspondiente precio, si es que está disponible en el catálogo:

```
/** Inner class OfferRequestsServer.
  This is the behaviour used by Book-seller agents to serve
  incoming requests for offer from buyer agents. If the
  requested book is in the local catalogue the seller agent
  replies with a PROPOSE message specifying the price.
  Otherwise a REFUSE message is sent back. */

private class OfferRequestsServer extends CyclicBehaviour {
  public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
      // Message received. Process it
      String title = msg.getContent();
      ACLMessage reply = msg.createReply();
      Integer price = (Integer) catalogue.get(title);
      if (price != null) {
        // The requested book is available for sale.
        // Reply with the price
        reply.setPerformative(ACLMessage.PROPOSE);
        reply.setContent(String.valueOf(price.intValue()));
      } else
      {
        // The requested book is NOT available for sale.
```

```
        reply.setPerformative(ACLMessage.REFUSE);
        reply.setContent( not-available );
      }
      myAgent.send(reply);
    }
  }
} // End of inner class OfferRequestsServer
```

Obsérvese que al ser un comportamiento cíclico y `receive()` un método no bloqueante, el agente estaría constantemente activando y desactivando el comportamiento mientras que no llegara un mensaje. Este proceso puede consumir una gran cantidad de CPU para hacer absolutamente nada útil. Lo que aconsejan los programadores de JADE para evitar esto es utilizar el método `block()` de la clase `Agent`. Con este método bloqueamos el comportamiento desde el cual se llama, sacándolo de la lista de comportamientos aun activos. Una vez que llega un mensaje a la cola del agente, todos los comportamientos bloqueados se vuelven a introducir al pool del agente correspondiente y vuelven a ser activos. El esquema sería

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

Un detalle que no hemos mencionado es el método `ACLMessage.createReply()`, que resulta muy útil ya que se encarga de crear un mensaje de respuesta correspondiente al mensaje instancia desde el que se invoca el método actualizando convenientemente el receptor del mensaje y los parámetros `conversation-id`, `reply-with` y `in-reply-to` cuando sea conveniente.

Un detalle que debemos tener en cuenta ahora nos va a servir para ir refinando la manera en la que los agentes manejan los mensajes que reciben ¿Os habeis preguntado ya cómo puede el agente estar seguro que el mensaje que obtiene de la cola es del tipo que desea? Obsérvese que en el comportamiento anterior, se escoge el primer mensaje de la cola y no se comprueba que sea del tipo `cfp` para así responder convenientemente a la petición de oferta con el precio del libro. La situación podría ser incluso más compleja ya que podríamos estar hablando de varios protocolos diferentes y que ambos usaran `cfp` por lo que no solamente habría que distinguir la performativa sino el protocolo para que el comportamiento correspondiente manejara el mensaje. Por otro lado, no basta con sacar el mensaje de la cola y comprobar que corresponde al que esperamos sino que además, si el mensaje no es el que buscamos, se debe conservar para que otros comportamientos lo manejen cuando proceda. Para manejar todo esto disponemos de la clase

```
jade.lang.acl.MessageTemplate
```

Una plantilla es una especie de patrón mediante la cual podemos especificar el tipo de mensaje que queremos recivir, como en el siguiente ejemplo:

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

El manejo de plantillas puede ser más complejo pudiéndose construir plantillas complejas en forma de expresiones lógicas con conectivas OR y AND, y especificándose matchings específicos para cualquier parámetro del mensaje. El correspondiente javadoc es bastante ilustrativo.

Para terminar con el envío y recepción de mensajes, vamos a ver una conversación compleja, correspondiente al comportamiento que implementa el envío de peticiones de ofertas, recoge todas las posibles ofertas y acepta una oferta determinada cuando proceda:

```
/** Inner class RequestPerformer. This is the behaviour
  used by Book-buyer agents to request seller agents the
  target book. */
private class RequestPerformer extends Behaviour {
  private AID bestSeller;
  // The agent who provides the best offer
  private int bestPrice;

  // The best offered price
  private int repliesCnt = 0;

  // The counter of replies from seller agents
  private MessageTemplate mt;

  // The template to receive replies
  private int step = 0;
  public void action() {
    switch (step) {
      case 0: // Send the cfp to all sellers
        ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
        for (int i = 0; i < sellerAgents.length; ++i) {
          cfp.addReceiver(sellerAgents[i]);
```

```
  }
  cfp.setContent(targetBookTitle);
  cfp.setConversationId(‘‘book-trade’’);
  cfp.setReplyWith(‘‘cfp’’ + System.currentTimeMillis());

  // Unique value
  myAgent.send(cfp);

  // Prepare the template to get proposals
  mt =
  MessageTemplate.and(MessageTemplate.MatchConversationId(‘‘book-trade’’),
  MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
  step = 1;
break;
case 1: // Receive all proposals/refusals from seller agents
  ACLMessage reply = myAgent.receive(mt);
  if (reply != null) { // Reply received
    if (reply.getPerformative() == ACLMessage.PROPOSE) {
      // This is an offer
      int price = Integer.parseInt(reply.getContent());
      if (bestSeller == null || price < bestPrice) {
        // This is the best offer at present
        bestPrice = price;
        bestSeller = reply.getSender();
      }
    }
    repliesCnt++;
    if (repliesCnt >= sellerAgents.length) {
      // We received all replies
      step = 2;
    }
  } else
  {
    block();
  }
break;
case 2: // Send the purchase order to the seller
        // that provided the best offer
  ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
  order.addReceiver(bestSeller);
  order.setContent(targetBookTitle);
  order.setConversationId(‘‘book-trade’’);
  order.setReplyWith(‘‘order + System.currentTimeMillis());
  myAgent.send(order);
  // Prepare the template to get the purchase order reply
  mt =
```

```
        MessageTemplate.and(MessageTemplate.MatchConversationId(``book-trade''),
      MessageTemplate.MatchInReplyTo(order.getReplyWith()));
      step = 3;
    break;
    case 3: // Receive the purchase order reply
      reply = myAgent.receive(mt);
      if (reply != null) { // Purchase order reply received
        if (reply.getPerformative() == ACLMessage.INFORM) {
          // Purchase successful. We can terminate
          System.out.println(targetBookTitle + `` successfully purchased.'');
          System.out.println(``Price = `` + bestPrice);
          myAgent.doDelete();
        }
        step = 4;
      } else {
        block();
      }
    break;
  }
}

public boolean done() {
  return ((step == 2 && bestSeller == null) || step == 4);
}
} // End of inner class RequestPerformer
```

## 5.5. Cómo localizar a los agentes vendedores

Hasta ahora hemos asumido que conocemos a los agentes vendedores, de tal forma que podemos acceder a sus nombres mediante un array fijo de valores. En sistemas abiertos de agentes en los que constantemente enstán entrando y de los que también salen constantemente, este esquema de funcionamiento no es posible. Por ello se hace necesario el mecanismo, que ya define FIPA tal y como hemos visto, para localizar determinados agentes por nombre o bien agentes que ofrecen determinados servicios que necesitamos.

Si hacemos memoria, recordaremos que en la arquitectura abstracta de FIPA hay un elemento opcional denominado facilitador de directorio (DF). La plataforma JADE incorpora este elemento en forma de clases Java para poder inscribir nuestros agentes en el sistema y para poder encontrar otros agentes que previamente se han suscrito a ellos y a sus servicios. Veámos el siguiente código, correspondiente a la forma de registrarse de un agente vendedor de libros en el ejemplo:

```
protected void setup() {
  ...
  // Register the book-selling service in the yellow pages
```

```
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription sd = new ServiceDescription();
  sd.setType(``book-selling'');
  sd.setName(``JADE-book-trading'');
  dfd.addServices(sd);
  try {
    DFService.register(this, dfd);
  } catch (FIPAException fe) {
    fe.printStackTrace();
  }
  ...
}
```

Obsérvese que el agente crea un objeto de la clase `DFAgentDescription` que responde a una entrada en el directorio que hace referencia a los datos de un agente. Un objeto de este tipo puede, a su vez, contener uno o más descripciones de servicios, objetos de la clase `ServiceDescription`. Una descripción de un servicio va a tener una serie de parámetros como el nombre y el tipo (además de todos los que consideremos oportuno añadir, dependiendo del dominio de aplicación). Así, se inscribe el agente comprador con un servicio del tipo de venta de libros. Téngase en cuenta que cuando el agente finaliza su ejecución hay de desubscribirlo tal que así:

```
protected void takeDown() { // Deregister from the yellow pages
  try {
    DFService.deregister(this);
  } catch (FIPAException fe) {
    fe.printStackTrace();
  } // Close the GUI
  myGui.dispose();
  // Printout a dismissal message
  System.out.println(``Seller-agent'' + getAID().getName() + `` terminating.'');
}
```

Si el agente vendedor debe inscribir sus servicios, el agente comprador debe ser capaz de encontrar agentes con servicios de venta de libros. Para ello podemos usar el método `DFService.search()` como en el fragmento de código del ejempl que sigue:

```
public class BookBuyerAgent extends Agent {
  // The title of the book to buy
  private String targetBookTitle;
  // The list of known seller
  agents private AID[] sellerAgents;
```

```
// Put agent initializations here
protected void setup() {
  // Printout a welcome message
  System.out.println(‘‘Hallo! Buyer-agent ‘‘ + getAID().getName() +
    ‘‘ is ready.’’);
  // Get the title of the book to buy as a start-up argument
  Object[] args = getArguments();
  if (args != null && args.length > 0) {
    targetBookTitle = (String) args[0];
    System.out.println(‘‘Trying to buy ‘‘ + targetBookTitle);

    // Add a TickerBehaviour that schedules a request to seller
    // agents every minute
    addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick() {
        // Update the list of seller agents
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType(‘‘book-selling’’);
        template.addServices(sd);
        try {
          DFAgentDescription[] result = DFService.search(myAgent,template);
          sellerAgents = new AID[result.length];
          for (int i = 0; i < result.length; ++i) {
            sellerAgents[i] = result.getName();
          }
        } catch (FIPAException fe) {
          fe.printStackTrace();
        }
        // Perform the request
        myAgent.addBehaviour(new RequestPerformer());
      } }
    );
  ...
```