Práctica 1

Introducción al desarrollo en la Plataforma JADE

David Oviedo
oviedo@dte.us.es

Plataforma JADE

- **J**ava **A**gent **DE**velopment framework
  - orientado al desarrollo de MAS de propósito general
  - integrado con el uso de un lenguaje ampliamente conocido (JAVA)
  - cumple con los estándares FIPA para la comunicación entre agentes e interplataforma
  - amplio uso tanto en entornos académicos como empresariales

IDE desarrollo

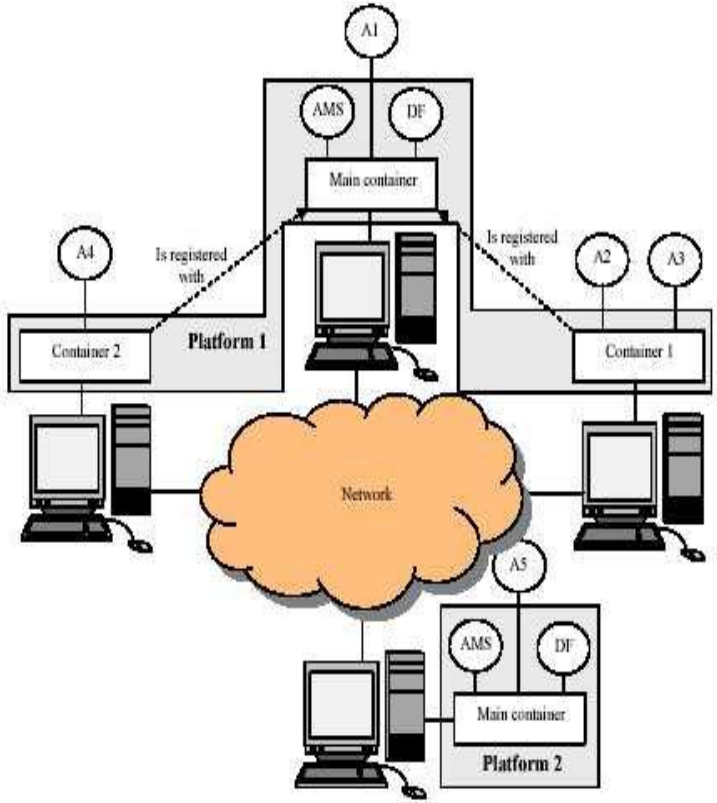
- Cualquiera para desarrollar en Java. En estas prácticas lo haremos con Eclipse.
- Los proyectos se enlazarán con las librerías de JADE (http://jade.tilab.com)
  - Trabajaremos con la última versión (jadeAll.zip_v.4.0.1) que ya se encuentra en c:/eclipse/jade
  - Crearemos un proyecto «Java Project» en c:/eclipse/jade/proyectos y enlazamos las librerias de JADE que se encuentran en c:/eclipse/jade/lib (jade.jar y commons-codec.jar)
  - A continuación crearemos una configuración de arranque:
    - Run->Run Configurations..->Java Applications -> Creamos una nueva configuración a la que llamaremos «JADE» y que simplemente llamará al main.class del default package.

Arranque de la plataforma

- Existen varias formas de arrancar la plataforma JADE (Por comandos, consola, desde una clase, remotamente, etc)
  - En este taller lo haremos desde la clase «main» de nuestro proyecto, indicándole una serie de parámetros.
  - La clase de arranque de la plataforma es «jade.Boot»
  - La opción «-gui» permite arrancar la interfaz de administración por defecto que trae JADE

Plataforma y Contenedores

- La plataforma JADE permite distribuir los agentes en distintas máquinas (hosts)
- JADE distribuye los agentes en los denominados "**Contenedores**", que es básicamente una máquina virtual de java.
  - En cada host, se ejecuta sólo una aplicación Java, y cada contenedor no es más que una JVM(Java Virtual Machine)
  - Permite simular por ejemplo en un mismo host varios contenedores (distintas máquinas virtuales)
  - JADE provee por defecto de un contenedor principal (obligatorio y sólo uno por plataforma), al cual se le asigna también un conjunto de agentes por defecto (los veremos más adelante) destinados a ofrecer los servicios de nombre, páginas amarillas y transporte de mensajes de la plataforma
  - El resto de contenedores que se definan, estarán interrelacionados con este contenedor principal (se registrarán en el mismo al comenzar la ejecución de la plataforma).



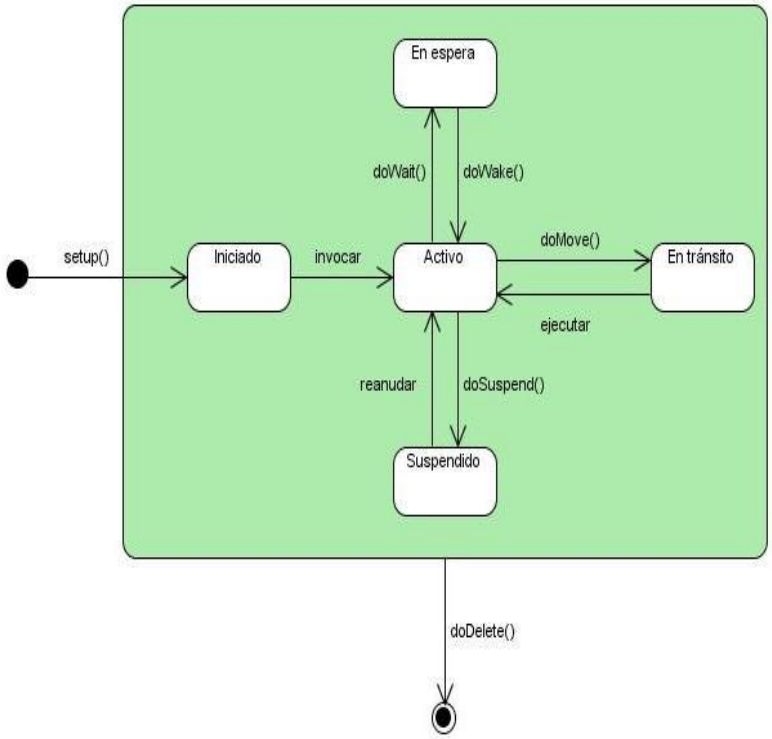
Agentes en JADE

- Programar un agente JADE consiste en:
  - Definir una clase Java que representa al agente (la cual debe heredar de la clase ***jade.core.Agent***).
  - Implementar los comportamientos que va a manisfestar.

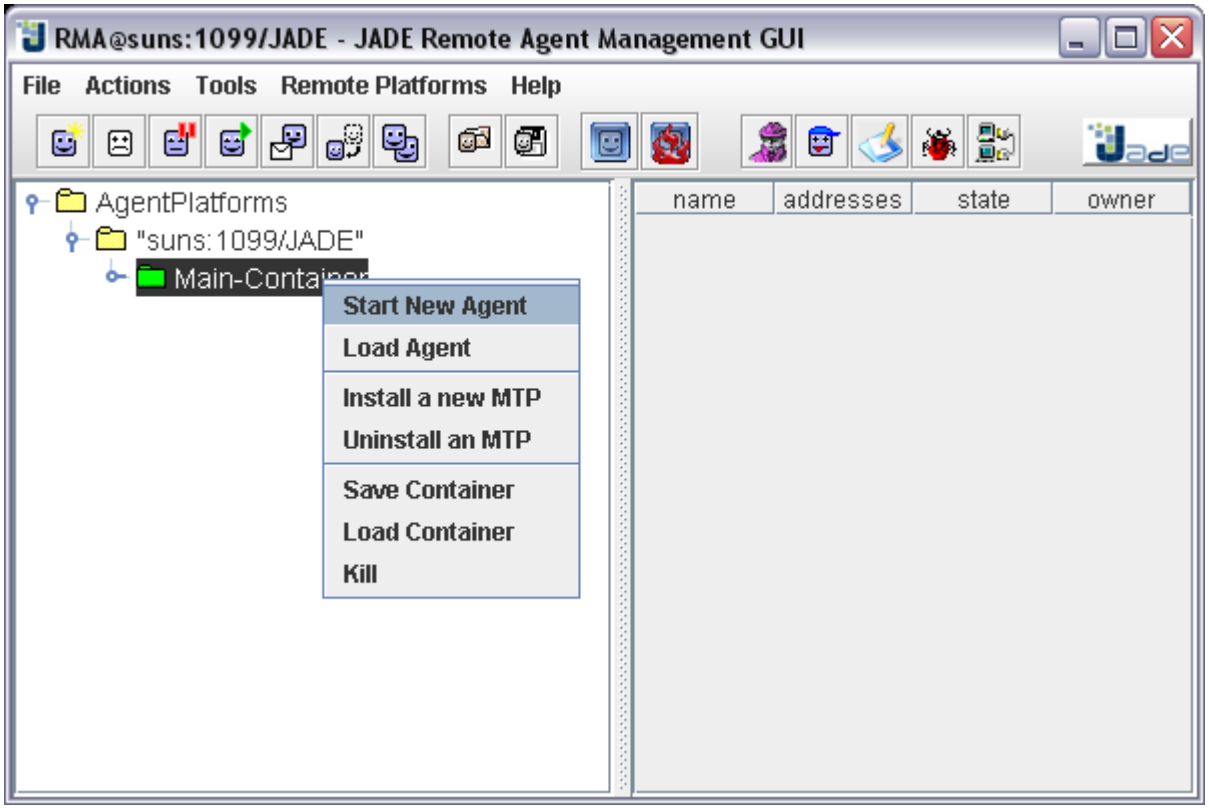
- Un agente JADE cumple las siguientes características:
  - Tiene un nombre único en el entorno de ejecución (**AID**).
  - Se implementa como un único hilo de ejecución.
  - Tiene un metodo de inicio (*setup*) y otro de fin (*takeDown*).
    - El método protegido *setup()* sirve para inicializar el agente incluyendo instrucciones que especificarán la ontología a utilizar y los comportamientos asociados al agente. Se invoca al comenzar la ejecución del agente.
    - El método protegido *takeDown()* sirve para liberar recursos antes de la eliminación del agente. Este método es invocado cuando se realiza una llamada al método *doDelete()*, que es el que realmente da por finalizada la ejecución del agente.

Ciclo de vida de un agente

- Un agente está sujeto a un ciclo de vida en el que se definen los estados en los cuales se puede encontrar el agente, así como los cambios que se pueden realizar entre los diferentes estados.
  - **Iniciado**: El objeto Agente está creado pero todavía no se ha registrado en el AMS, no tiene nombre ni dirección y tampoco se puede comunicar con otros agentes.
  - **Activo**: El Agente está registrado en el AMS, tiene un nombre, una dirección y puede acceder a todas las opciones de JADE.
  - **Suspendido**: El Agente está parado. Su hilo de ejecución está detenido y no ejecuta ningún Comportamiento.
  - **En espera**: El Agente está bloqueado esperando por algo. Su hilo de ejecución está dormido en un monitor de java y se despertará cuando se cumpl una cierta condición (cuando reciba un mensaje)
  - **Desconocido**: El Agente ha sido eliminado. El hilo de ejecución ha terminado y se ha eliminado del registro del AMS.
  - **Tránsito**: Un Agente móvil entra en este estado mientras está migrando a una nueva localización. El sistema sigue guardando los mensajes en el buffer hasta que el agente vuelve a estar activo.



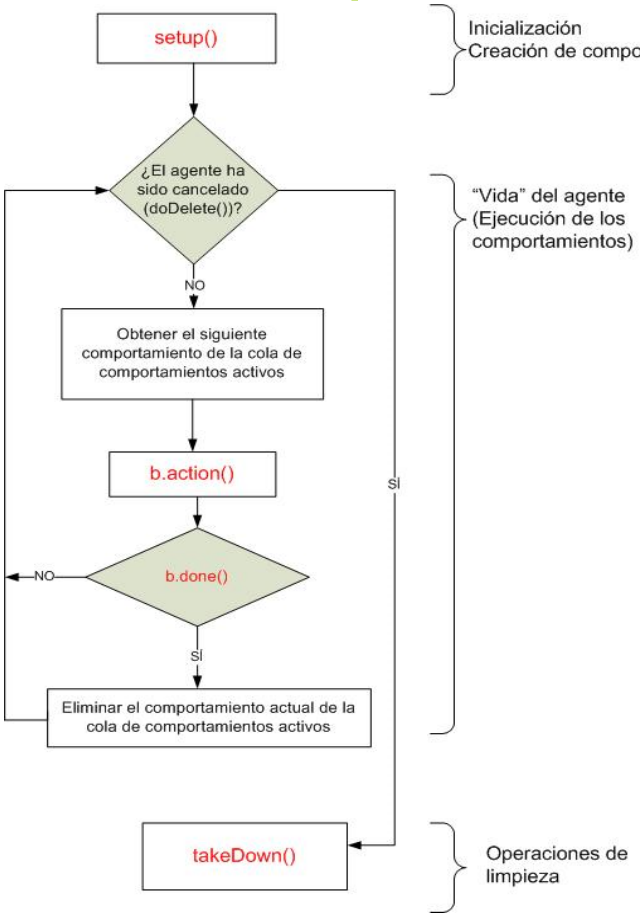
GUI y Recursos de JADE



Comportamientos de un agente

- En la implementación de un agente se define una clase interna por cada uno de los comportamientos asociados al agente. Estos comportamientos se utilizan básicamente para el envío y recepción de mensajes, aunque también se pueden utilizar para realizar otras tareas.
  - Para implementar comportamientos, debemos importar: ***jade.core.behaviours.****
  - Existen diferentes tipos de comportamiento:
    - Comportamiento genérico -> ***Behaviour***
    - Comportamientos simples -> ***OneShotBehaviour y CyclicBehaviour***
    - Comportamientos compuestos -> ***SequentialBehaviour, ParallelBehaviour y FSMBehavior***
  - La clase interna que define un comportamiento debe heredar de una de las anteriores.
  - Para registrar ó eliminar un comportamiento en un agente, se debe realizar mediante los siguientes métodos:
    - **addBehaviour(Behaviour)** y **removeBehaviour(Behaviour)**

Ejecución de los comportamientos



- Cada agente tiene un planificador o scheduler de comportamientos.
- Los comportamientos no son ejecutados concurrentemente.
- El funcionamiento de los comportamientos está implementado a 2 niveles:
  - Una cola circular de los comportamientos activos
  - Una cola con los comportamientos bloqueados
- Un comportamiento puede bloquearse (**block()**) cuando el método **action()** termina, entonces éste se coloca en la cola de los comportamientos bloqueados. Cuando este comportamiento se desbloquea (llegada de un mensaje, condición, etc.) se saca de la cola de bloqueados y se coloca al final de la cola de comportamientos activos.

Métodos genéricos de un comportamiento

- Toda clase que herede de Behaviour deberá implementar al menos:
  - el método **action()** :Este método define la acción a ser ejecutada cuando se ejecute el comportamiento. Debe incluir el código de las acciones a realizar cuando se ejecute el comportamiento.
    - Es invocado cuando se produce el evento asociado al comportamiento.
    - Es recomendable que los métodos **action()** no tengan un tiempo de ejecución alto ya que mientras que se ejecutan no pueden ser interrumpidos por otro comportamiento.
  - el método **done()**: Es invocado cuando finaliza la ejecución del método **action()**.
    - Este método determina si el comportamiento ha sido completado o no. Devuelve un booleano (*true* si ha terminado o *false* en caso contrario).
    - Si el comportamiento ha finalizado, éste se elimina de la cola de comportamientos activos.
    - Se puede utilizar una marca que se activa cuando se quiere que finalice el comportamiento (se evalúa su valor en el método done()).
- Un comportamiento también puede ser bloqueado utilizando el método **block()**. Este método permite bloquear un comportamiento hasta que algún acontecimiento ocurra (típicamente, hasta que un mensaje llegue). Cuando el método **action()** termina, el método **block()** coloca el comportamiento en la cola de comportamientos bloqueados.
  - Debe tenerse en cuenta que el método **block()** no es como el método sleep() de los Threads. El método **block()** no para la ejecución del comportamiento sino que espera a que finalice el método **action().** Una vez finalizado, si el comportamiento no termina, éste pasa a la lista de comportamientos bloqueados durante el tiempo que indique el método **block()** o hasta que se reciba un nuevo mensaje.

Agentes de la Plataforma JADE

- JADE provee un conjunto de agentes destinados a resolver y dar soporte a funcionalidades básicas de la plataforma:
  - AMS (Agent Management System): Agente encargado de ofrecer los servicios de páginas blancas(Directorio de agentes disponibles en la plataforma) , garantizando que cada agente dentro de la plataforma tenga un nombre único (AID: Agent Identifier) y actualizando el estado de los mismos (Disponible, No disponible, Saturado, etc.) de cada uno de los agentes
  - DF (Directory Facilitator): Agente encargado de ofrecer los servicios de páginas amarillas (Directorio de los servicios ofrecidos por los agentes disponibles en la plataforma). Si un agente quiere que un servicio propio esté disponible para el conjunto de agentes de la plataforma, debe registrarlo en el DF.
  - RMA (Remote Monitoring Agent): Agente encargado de monitorear el estado completo de la plataforma (agentes y contenedores de agentes) ofreciendo un conjunto de herramientas destinadas a la depuración y desarrollo de la misma:
  - SF (Sniffer Agent): Seguimiento/Tracking de mensajes intercambiados entre agentes.
  - DA (Dummy Agent): Inspección del contenido de los mensajes intercambiados entre agentes.

Agente DF

- Los agentes interactúan con el DF intercambiando mensajes ACL usando el lenguaje SL y la ontología FIPA-agent-management.

- Publicar servicios:
  - El agente debe proporcionar al DF una descripción, incluyendo su AID, los protocolos, lenguajes y ontologías que el resto de agentes necesitan conocer para interactuar con él; y la lista de servicios publicados.
  - Para cada servicio se proporciona una descripción, incluyendo: tipo de servicio, nombre, protocolos, lenguajes y ontologías; y una serie de propiedades específicas del servicio.
  - Antes de finalizar su ejecución el agente debe eliminar del DF sus servicios.

- Busqueda de servicios:
  - Un agente que busca servicios debe proporcionar una plantilla de descripción de la clase DFAgentDescription.
  - El resultado de la búsqueda es la lista de todas las descripciones que encajan con la plantilla proporcionada (los campos especificados en la plantilla están presentes en la descripción con los mismos valores, los campos no cubiertos no se comprueban).

Comunicaciones entre agentes

- Para que los agentes se puedan comunicar deben usar el mismo lenguaje de comunicación, un mismo lenguaje de contenido y una misma ontología
  - Lenguaje de comunicación (ACL – Agent Communication Language): Define el tipo de mensajes y el protocolo de comunicaciñon.
  - Lenguaje de contenido: representación interna del contenido de los mensajes ACL (modo de encapsular o codificar la semántica u objetos de la ontología dentro de mensajes ACL)
  - Ontologías: Definen la semántica de los mensajes que se intercambian.
- En el caso de JADE:
  - el lenguaje de comunicación entre agentes es FIPA-ACL.
  - soporte (codecs), para dos lenguajes de contenido: SL (codifica las expresiones como string) y LEAP (byte-encoded)

FIPA-ACL

- Un mensaje FIPA-ACL va a contener un conjunto de parámetros que definen varios aspectos del mensaje:
  - Performativa: indica el tipo de acto comunicativo del mensaje. Existen varios tipos para JADE: **not-understood**, **inform**, **query**, …
- Un mensaje en JADE es implementado mediante la creación de un objeto de la clase ***jade.lang.acl.ACLMessage***, la cual provee métodos ***get*** y ***set*** para el manejo de todos los parámetros vistos anteriormente. Ejemplo:

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Table 1: FIPA ACL Message Parameters

```
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);
```

Envío/recepción de mensajes

- Mecanismo: paso asíncrono de mensajes
- Cada agente tiene una cola de mensajes entrantes. La lectura efectiva de los mensajes es a voluntad del agente
- Un agente puede:
  - Leer el primer mensaje en la cola
  - Leer el primer mensaje que satisfaga un requisito
- La cola de mensajes es única para cada agente y, por lo tanto, es compartida por todos los comportamientos
- Cada vez que se coloca un mensaje en la cola el agente receptor es avisado
- Un comportamiento puede ser bloqueado en espera de la recepción de un mensaje: sincronización
- Para recibir mensajes en un agente se usará el método **receive()** de la clase Agent. Este método obtiene el primer mensaje de la cola de mensajes y lo devuelve (devuelve null si la cola está vacía).

```
ACLMessage mensaje = receive();
```

- Para enviar un mensaje desde un agente se llama al método **send()** de la clase Agent. El método send() recibe como parámetro un ACLMessage, añade el valor oportuno al campo sender (remitente) y envía el mensaje a los destinatarios

Métodos de ACLMessage

A continuación se muestran algunos de los métodos más importantes de la clase ACLMessage. Para consultar más detenidamente todos los métodos de dicha clase se puede consultar su documentación en la API de JADE

- **setPerformative**(int p): toma como parámetro una constante representativa de un tipo de acción performativa y la establece como performativa del mensaje.Por ejemplo, para hacer que el mensaje msg sea de tipo agree bastará con escribir:

```
msg.setPerformative(ACLMessage.AGREE);
```

- **getPerformative**(): devuelve un entero equivalente a la constante que representa a la performativa del mensaje
- **createReply**(): crea un mensaje de respuesta para el mensaje sobre el que es aplicado, poniendo los valores oportunos en campos como receiver, conversation-id, etc.
- **addReceiver**(AID): toma como parámetro un AID y lo añade a la lista de receptores
- **getAllReceiver**(): devuelve un iterador sobre la lista de receptores.
- **setContent**(String): recibe como parámetro una cadena y la pone como contenido del mensaje
- **getContent**(): devuelve una cadena con el contenido del mensaje

Ejemplo de intercambio de mensajes entre dos agentes:

- Necesario dos agentes, uno emisor y otro receptor
- El intercambio puede realizarse mediante plantillas(sincronización) o sin ellas
- Ver posibilidades de depuración que ofrece la GUI de JADE