Laboratorio

Plataforma de Agentes Jade

Ulises Cortés

Ignasi Gómez-Sebastià

Sergio Álvarez

SID2018

Agente: Clase

- Una clase de agente se crea extendiendo la clase *jade.core.Agent*
  - Se redefine el método *setup()*
- Cada instancia de agente se identifica con un *AID*
  - Clase *jade.core.AID*
  - Método *getAID()* para obtener el AID del agente

```
import jade.core.Agent;

public class HalloWorldAgent extends Agent {

  protected void setup() {
    System.out.println("Hallo World! my name is "+getAID().getName());
  }
}
```

Agente: Nombre

- El nombre del agente es de la forma
  - *nombre-agente@nombre-plataforma*
  - El nombre del agente debe ser único localmente, el nombre completo debe ser único globalmente
- Se puede especificar el nombre de la plataforma al arrancarla usando la opción *–name*
- Dentro de una plataforma nos referimos al agente usando únicamente su nombre local
  - `AID id = new AID(localname, AID.ISLOCALNAME);`
  - `AID id = new AID(name, AID.ISGUID);`

Agente: Parámetros

- Paso de parámetros a un agente al arrancarlo

```
java jade.Boot .... a:myPackage.MyAgent(arg1 arg2)
```

- Recuperamos los parámetros usando el método *getArguments()*

```
protected void setup() {
  System.out.println("Hallo World! my name is "+getAID().getName());
  Object[] args = getArguments();
  if (args != null) {
    System.out.println("My arguments are:");
    for (int i = 0; i < args.length; ++i) {
      System.out.println("- "+args[i]);
    }
  }
}
```

Agente: Clase

- HelloWorldAgent.java

```
package org.upc.edu.Behaviours;



/**
 *
 * @author igomez
 */
import jade.core.Agent;

public class HelloWorldAgent extends Agent
{



    protected void setup()
    {

    String arguments = "";

    Object[] args = getArguments();
    if (args != null) {
    for (int i = 0; i < args.length; ++i)
    {
        arguments = arguments + " - " + args[i];
    }
    }


    System.out.println("Hallo World! my name is "+getAID().getName() + " and my arguments are " + arguments);
    }
}
```

Agente: Clase

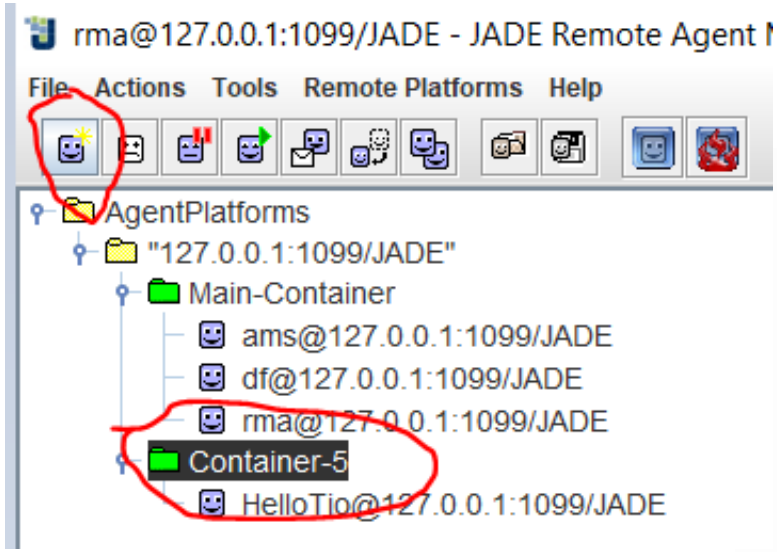
```
--Hello World Agent Windows
start java -cp ./lib/jade.jar;./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -gui
start java -cp ./lib/jade.jar;./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -container
HelloTio:org.upc.edu.Behaviours.HelloWorldAgent("Que","pasa","tio")

--Hello World Agent Linux
java -cp ./lib/jade.jar:./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -gui &
java -cp ./lib/jade.jar:./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -container
HelloTio:org.upc.edu.Behaviours.HelloWorldAgent\("Que","pasa","tio"\) &
```

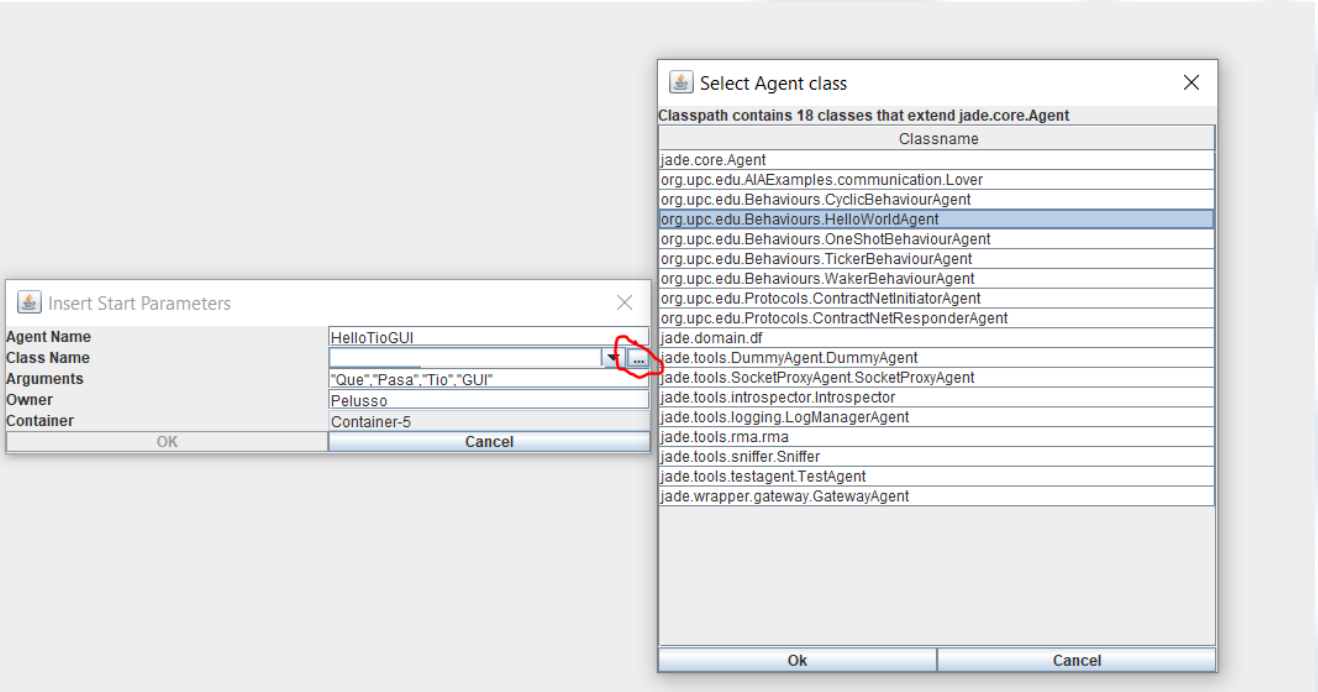
```
mar 13, 2019 11:02:50 AM jade.core.Runtime beginContainer
INFO: ----------------------------------
    This is JADE snapshot - revision 6357 of 2010/07/06 16:27:34
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
Retrieving CommandDispatcher for platform null
mar 13, 2019 11:02:50 AM jade.imtp.leap.LEAPIMTPManager initialize
INFO: Listening for intra-platform commands on address:
- jicp://127.0.0.1:51000

mar 13, 2019 11:02:51 AM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
mar 13, 2019 11:02:51 AM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
mar 13, 2019 11:02:51 AM jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
mar 13, 2019 11:02:51 AM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
mar 13, 2019 11:02:51 AM jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
Hallo World! my name is HelloTio@127.0.0.1:1099/JADE and my arguments are  - Que - pasa - tio
mar 13, 2019 11:02:51 AM jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Container-5@127.0.0.1 is ready.
----------------------------------------------
```

Agente: Clase

```
--Hello World Agent Windows
start java -cp ./lib/jade.jar;./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -gui
start java -cp ./lib/jade.jar;./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -container
HelloTio:org.upc.edu.Behaviours.HelloWorldAgent("Que","pasa","tio")

--Hello World Agent Linux
java -cp ./lib/jade.jar:./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -gui &
java -cp ./lib/jade.jar:./dist/JadeApplication.jar jade.Boot -local-host 127.0.0.1 -container
HelloTio:org.upc.edu.Behaviours.HelloWorldAgent\("Que","pasa","tio"\) &
```

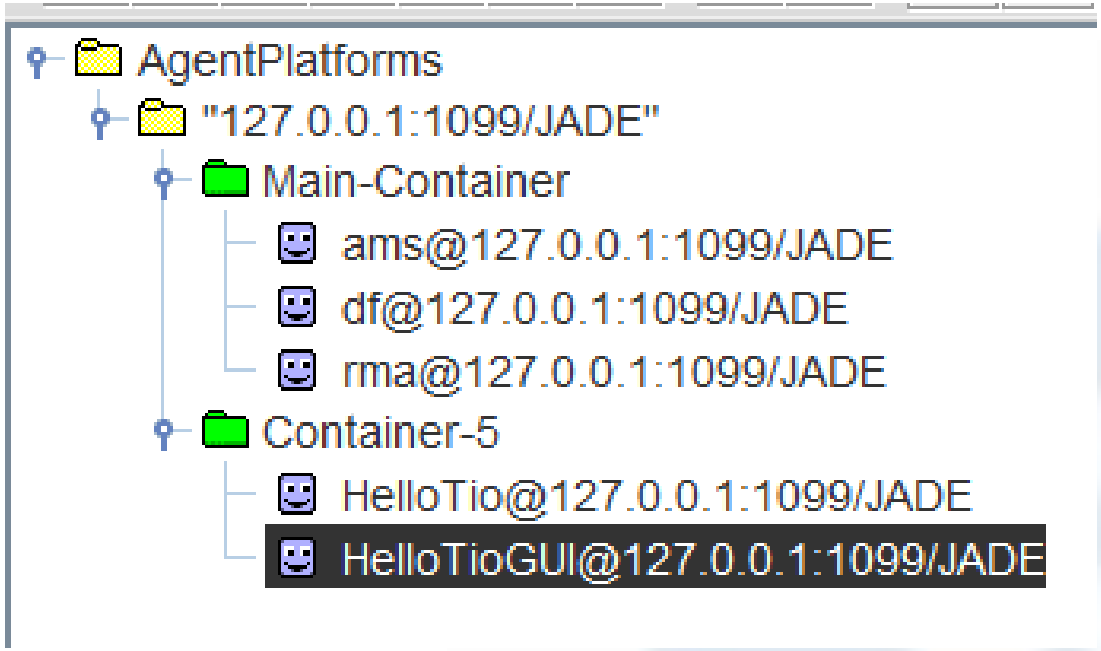
Agente: Clase





Agente: Clase

```
----------------------------------------
Hallo World! my name is HelloTioGUI@127.0.0.1:1099/JADE and my arguments are  - "Que" - "Pasa" - "Tio" - "GUI"
```



Agente: Clase

- Ejercicios
  - Descargar plataforma desde el racó
  - Abrir con editor
    - Netbeans, Eclipse, etc.
  - Plataforma desde consola
    - Comprobar logs
  - Arrancar agente HelloWorldAgent desde consola
    - Comprobar logs
  - Arrancar agente HelloWorldAgent desde GUI
    - Comprobar logs
    - Se han pasado igual los parámetros?

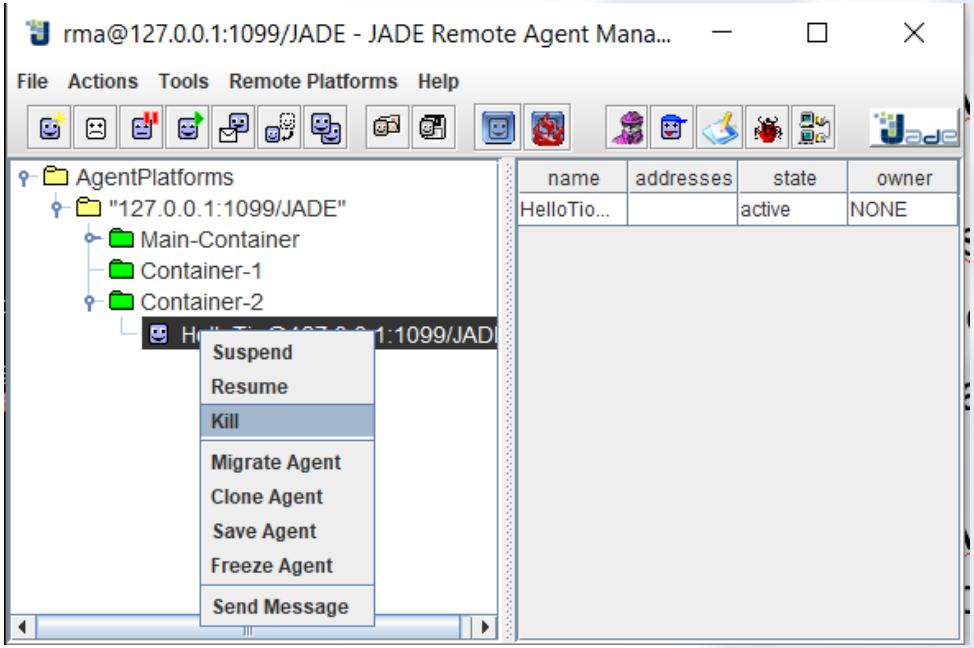
Agente: Defunción

- El agente termina su ejecución cuando se llama su método *doDelete()*
- Se invoca el método *takeDown()* para realizar operaciones de limpieza
  - Cerrar ficheros y conexiones

```
protected void setup() {
  System.out.println("Hallo World! my name is "+getAID().getName());
  Object[] args = getArguments();
  if (args != null) {
    System.out.println("My arguments are:");
    for (int i = 0; i < args.length; ++i) {
      System.out.println("- "+args[i]);
    }
  }
  doDelete();
}

protected void takeDown() {
  System.out.println("Bye...");
}
```

Agente: Clase

- Ejercicios
  - Ampliar el agente HelloWorldAgent para añadir operación takeDown
    - Mostrar mensaje en los logs en la operación
  - Arrancar el agente y matarlo desde la GUI
    - Comprobar logs y ver que la operación es ejecuta
  - Ampliar el agente HelloWorldAgent para que se mate tras mostrar el mensaje con nombres y argumentos



Agente: Behaviours

- Los agentes realizan sus tareas a partir de *behaviours*
  - Extensión de la clase *jade.core.behaviours.Behaviour*
- Creamos nuestra clase behaviour y la añadimos al agente usando la función *addBehaviour()*
  - Podemos compartir behaviours entre agentes
- Cada behaviour debe implementar:
  - Método *void action()*
    - Que hace el behaviour
  - Método *boolean done()*
    - Devuelve cierto si el behaviour ha finalizado

Agente: Behaviours

```
public class CyclicBehaviourAgent extends Agent
{

    public class HelloWorldCyclicBehaviour extends CyclicBehaviour
    {
        String message;
        int count_chocula;

        public HelloWorldCyclicBehaviour()
        {

        }

        public void onStart()
        {
            this.message = "Agent " + myAgent +" with HelloWorldCyclicBehaviour in action!!" + count_chocula;
            count_chocula = 0;
        }

        public int onEnd()
        {
            System.out.println("I have done " + count_chocula + " iterations");
            return count_chocula;
        }

        public void action()
        {
            System.out.println(this.message + count_chocula);
            ++count_chocula;
        }

    }
    protected void setup()
    {

     HelloWorldCyclicBehaviour b = new HelloWorldCyclicBehaviour();
     this.addBehaviour(b);
    }
}
```

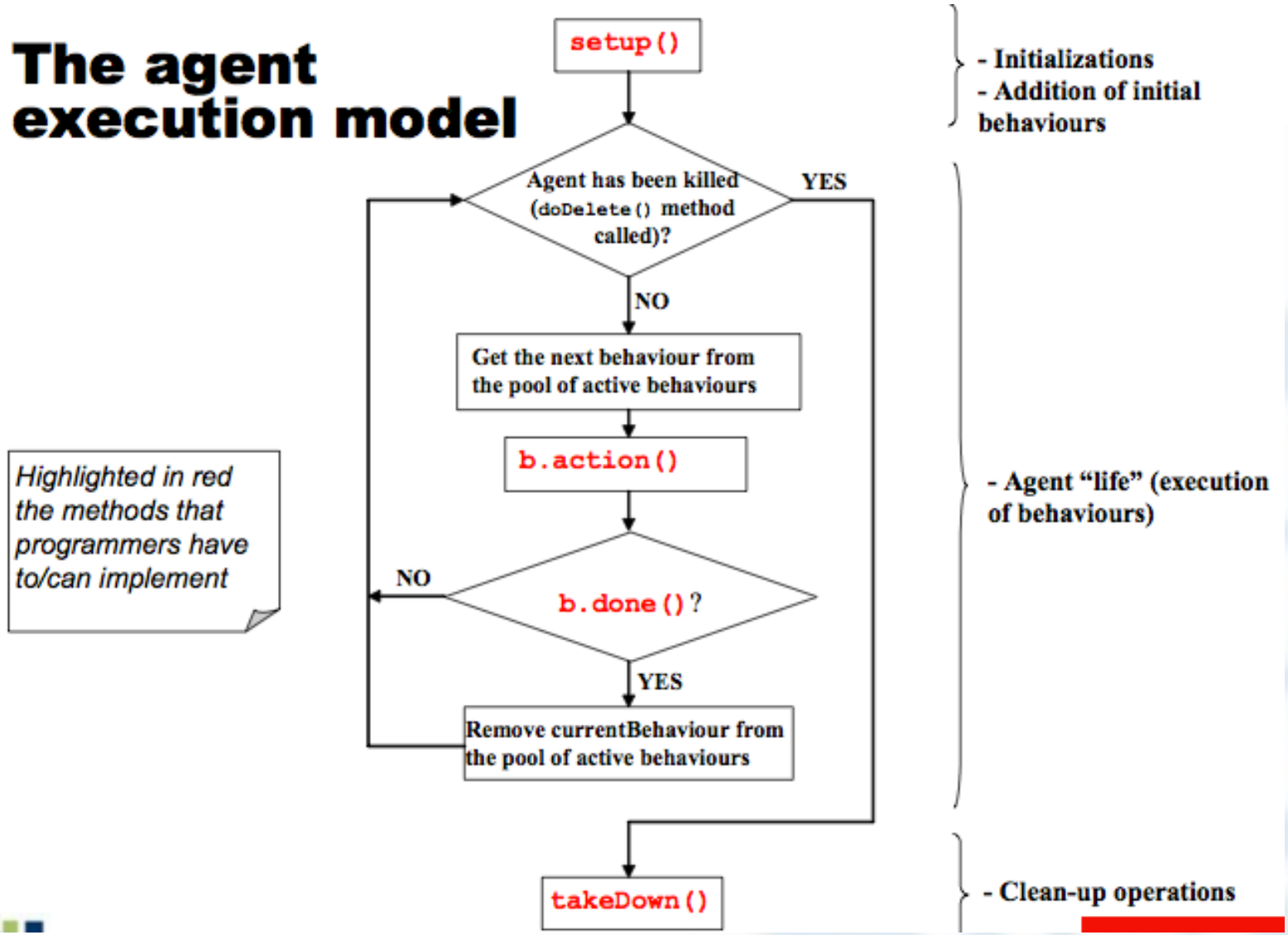
Agente: Behaviours

- Ejercicios
  - Ejecutar el agente CyclicBehaviourAgent desde consola
  - Comprobar los logs
  - Matar el agente desde GUI
  - Extender el behaviour con el método done
    - El behaviour termina al escribir 10 veces
  - Ejecutar el agente CyclicBehaviourAgent desde consola y mirar cambios en los logs

Agente: Concurrencia

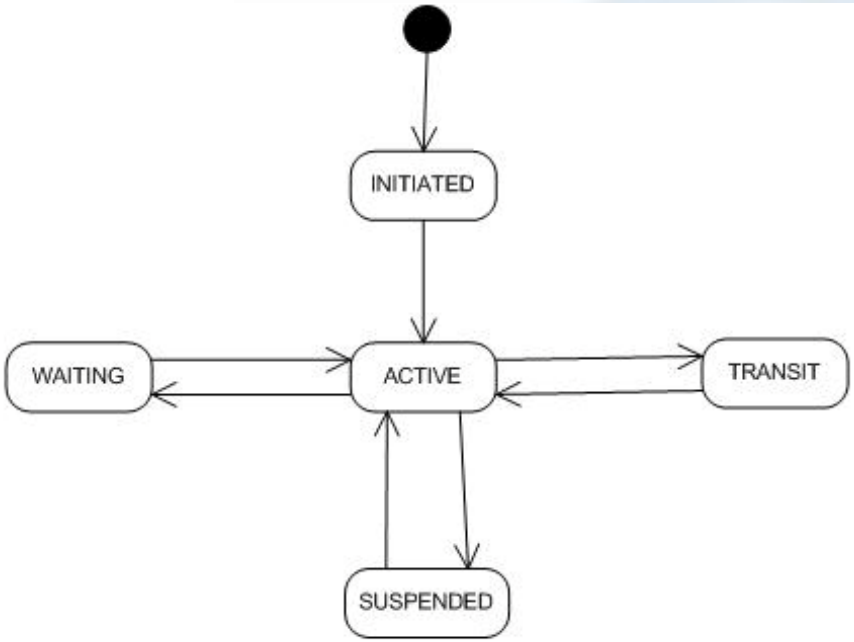
- Un agente puede tener varios comportamientos en paralelo
- El planificado no es *preemptive*
  - Se debe esperar a que termine un behaviour para seleccionar el siguiente
    - Método *action()* retorna
  - Todo ocurre dentro del mismo thread de Java
  - No es determinista a la hora de seleccionar un behaviour si hay varios disponibles

Agente: Ciclo de ejecución



Agente: Ciclo de ejecución

State	Description
Initiated	The agent object is built, but hasn't registered itself with AMS, has neither name nor an addressand can't communicate with other agents.
Active	The agent object is registered with AMS, has regular name and address, can access all the various JADE features.
Suspended	The agent object is currently stopped. Its internal thread is suspended and no agent behaviour is being executed.
Waiting	The agent is blocked, waiting for somethink. Its internal thread is sleeping on a JAVA monitor and will wake up when some conditions are met (for example when message arrives).
Deleted	The agent is definitely dead. The intternal thread has terminated its execution and the agent is no more registered with AMS.
Transit	The mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.



Agente: Tipos de behaviour I

- One shot
  - jade.core.behaviours.OneShotBehaviour
  - Action() se ejecuta una vez
  - Done() devuelve cierto
- Cyclic
  - jade.core.behaviours.CyclicBehaviour
  - Action() se ejecuta multiples veces
  - Done() devuelve falso
- Complex
  - Basados en máquinas de estados finitas

Agente: Tipos de behaviour II

- Waker
  - Espera un time-out y ejecuta método *onWake()*
  - El behaviour se completa
  - Despertador chino
- Ticker
  - Se ejecuta método *onTick()* periodicamente
  - Se para al ejecutar método *stop()*

Agente: Tipos de behaviour II

```
public class CyclicBehaviourAgent extends Agent
{

    public class HelloWorldCyclicBehaviour extends CyclicBehaviour
    {
        String message;
        int count_chocula;

        public HelloWorldCyclicBehaviour()
        {

        }

        public void onStart()
        {
            this.message = "Agent " + myAgent +" with HelloWorldCyclicBehaviour in action!!" + count_chocula;
            count_chocula = 0;
        }

        public int onEnd()
        {
            System.out.println("I have done " + count_chocula + " iterations");
            return count_chocula;
        }

        public void action()
        {
            System.out.println(this.message + count_chocula);
            ++count_chocula;
        }

    }
    protected void setup()
    {

     HelloWorldCyclicBehaviour b = new HelloWorldCyclicBehaviour();
     this.addBehaviour(b);
    }
}
```

Agente: Tipos de behaviour II

```
public class OneShotBehaviourAgent extends Agent
{

     public class HelloWorldOneShotBehaviour extends OneShotBehaviour
    {
        String message;
        int count_chocula;

        public HelloWorldOneShotBehaviour()
        {

        }

        public void onStart()
        {
            this.message = "Agent " + myAgent +" with HelloWorldOneShotBehaviour in action!!";
            count_chocula = 0;
        }

        public int onEnd()
        {
            System.out.println("I have done " + count_chocula + " iterations");
            return count_chocula;
        }

        public void action()
        {
            System.out.println(this.message + count_chocula);
        }

    }

     protected void setup()
    {

      HelloWorldOneShotBehaviour b = new HelloWorldOneShotBehaviour();
      this.addBehaviour(b);
    }
}
```

Agente: Tipos de behaviour II

```
public class TickerBehaviourAgent extends Agent
{
    public class HelloWorldTickerBehaviour extends TickerBehaviour
    {
        String message;
        int count_chocula;

        public HelloWorldTickerBehaviour(Agent a, long period)
        {
            super(a,period);
        }

         public void onStart()
        {
            this.message = "Agent " + myAgent +" with HelloWorldTickerBehaviour in action!!";
            count_chocula = 0;
        }

        public int onEnd()
        {
            System.out.println("I have done " + count_chocula + " iterations");
            return count_chocula;
        }

        public void onTick()
        {
            System.out.println(this.message + count_chocula);
            ++count_chocula;
        }

    }

    protected void setup()
    {

     HelloWorldTickerBehaviour b = new HelloWorldTickerBehaviour(this, 3000);;
     this.addBehaviour(b);
    }
```

Agente: Tipos de behaviour II

```
public class WakerBehaviourAgent extends Agent
{

    public class HelloWorldWakerBehaviour extends WakerBehaviour
    {
        String message;
        int count_chocula;

        public HelloWorldWakerBehaviour(Agent a, long timeout)
        {
            super(a,timeout);
        }

        public void onStart()
        {
            this.message = "Agent " + myAgent +" with HelloWorldWakerBehaviour in action!!";
            count_chocula = 0;
        }

        public int onEnd()
        {
            System.out.println("I have done " + count_chocula + " iterations");
            return count_chocula;
        }

        public void onWake()
        {
            System.out.println(this.message + count_chocula);
            ++count_chocula;
        }

    }

     protected void setup()
    {

     HelloWorldWakerBehaviour b = new HelloWorldWakerBehaviour(this, 3000);
     this.addBehaviour(b);
    }
```

Agente: Tipos de behaviour II

- Ejercicios
  - Ejecutar los siguientes tipos de agentes y comprobar su comportamiento
    - OneShot
    - Ticker
    - Waker
  - Encajan con la descripción realizada?

Agente: Métodos

- onStart()
  - Se ejecuta una vez antes de ejecutar método *action()* por primera vez
  - Prepara la ejecución (abrir ficheros y conexiones)
- onEnd()
  - Se ejecuta una vez después de que onEnd() devuelva cierto
  - Limpia la ejecución (cerrar ficheros y conexiones)
- removeBehaviour()
  - Elimina un behaviour dispnonible de la lista
  - No llama a onEnd()
- Si el agente no tiene behaviours disponibles está IDLE y el thread en sleep

Interacciones entre agentes

- Las interacciones entre agentes son inevitables
  - Para alcanzar los objetivos
    - Propios
    - Comunes a un grupo de agentes
  - Para tratar interdependencias con otros actores
- Las interacciones ocurren en el *Knowledge level*
  - Qué objetivos persigue el agente
  - Bajo qué condiciones (cuándo)
  - Quién ejecuta cada acción
- Flexibilidad para actuar autónomamente y comunicarse con el resto de agentes
  - E.g., programas síncronos

Interacciones entre agentes

- Motivación:
  - El intercambio de conocimiento es un signo de inteligencia
  - El intercambio de conocimiento requiere comunicación
- La idea principal detrás del paradigma de los sistemas multi-agente se basa en la existencia de entidades de software distribuidas y heterogéneas que se comunican entre ellas
- El hecho de que los agentes sean diversos y heterogéneos implica que necesitamos un lenguaje común

¿Por qué se comunican?

- Entendimiento mutuo
  - Traducir entre lenguajes de representación
  - Compartir el contenido semántico del lenguaje
- Componentes de la comunicación a acordar:
  - Protocolo de interacción
    - Cómo se estructuran las conversaciones o diálogos
  - Lenguaje de comunicación
    - Cual es el significado de cada mensaje

¿Por qué se comunican?

- Protocolo de transporte
  - Detalles técnicos sobre como los mensajes son enviados y recibidos por los agentes
  - Oculto para los desarrolladores por parte de la plataforma JADE
- Detalles técnicos sobre la arquitectura y el middleware
  - Ya solucionado por los estándares FIPA

Los agentes como base de conocimiento

- Podemos considerar a los agentes inteligentes como bases (virtuales) de conocimiento
- Tres niveles de representación:
  - Lenguaje o formalismo que permita representar el conocimiento del dominio
    - Ontología
  - Lenguaje que permita expresar proposiciones
    - Para habilitar el intercambio de información
    - *Content Language* implícito en los mensajes
  - Lenguaje que permita expresar actitudes sobre esas proposiciones
    - ACL (Agent Communication Language)

Niveles en la comunicación

- Semántica del mensaje
  - Que quiere decir mensaje
  - Subdividido en:
    - Tipo de mensaje: Intencionalidad
    - Contenido del mensaje: Contiene la información
    - Ontología: Conceptos a los que se refiere el mensaje

Niveles en la comunicación

- Sintaxis del mensaje:
  - Como se expresa cada mensaje
    - Estructura: ACL (Agent Comunication Language)
    - Contenido: Content language
- Protocolo de interacción
  - Forma en que se estructuran las conversaciones o diálogos
    - Protocolos de agentes FIPA
- Protocolo de transporte
  - Forma a nivel técnico en que los mensajes se envían y reciben
    - Oculto por la plataforma

Semántica del mensaje

- La mayoría de los enfoques para modelar formalmente la comunicación entre agentes se inspiran en el *Speech Act Theory*
  - *How to do things with words. Austin 1962*
- Las *Speech Act Theory* son un conjunto de teorías pragmáticas sobre el lenguaje
  - Tratan sobre el uso del lenguaje
  - Abordan el uso del lenguaje en el día a día
  - Estudian el uso del lenguaje como forma de alcanzar objetivos e intenciones

Semántica del mensaje

- Idea principal detrás de la *Speech Act Theory*
  - Algunos actos de comunicación son equivalentes a acciones físicas que cambian el estado del mundo
    - Declaración formal de guerra
    - 'Os declaro marido y mujer'
    - Silbido para indicar que es penalti
- En general, todos nuestros actos de comunicación se realizan con la intención de alcanzar un objetivo o intención
- La *Speech Act Theory* explica como los actos de comunicación se usan para alcanzar un objetivo

Semántica del mensaje

- Esto es un proyecto Netbeans
  - Sugiero que uséis Netbeans
  - Os obligo a usar Nebteans
  - Os pido que useis Netbeans
  - Os pregunto si soléis usar Netbeans
  - Os anuncio que yo suelo usar Netbeans

Aspectos del Speech Act

- Tres aspectos del speech act:
  - Locución o acto locutorio
    - Lo que se dice (Uso Netbeans)
  - Ilocución o acto ilocutorio
    - Lo que no se dice explícitamente, pero se dice
    - Sugerencia, petición, información
    - La fuerza ilocutoria se aplica al contenido
  - Perlocución o acto perlocutorio
    - El efecto que provoca la comunicación en el receptor
    - La fuerza perlocutoria está relacionada con las intenciones del agente

Componentes del speech act

- Verbo performativo
- Contenido proposicional
- Ejemplo I
  - Performativa: Petición
  - Contenido: “La puerta está abierta”
  - Speech act: Por favor, cierra la puerta
- Ejemplo II
  - Performativa: Información
  - Contenido: “La puerta está abierta”
  - Speech act: Te informo de que la puerta está abierta

Semántica del speech act

- Dado un conjunto de ilocuciones
  - (request, agent1, agent2)
  - (inform, agent1, agent2)
  - (ask, agent1, agent2)
- Especifican las condiciones de éxito para cada ilocución
  - Cuales son las condiciones necesarias y suficientes que deben darse para que el agent1 pueda considerar que el request(ask, inform, etc.) hacia el agent2 ha sido exitosa

Sintaxis del Speech Act

- Dos enfoques
  - Procedural
    - Intercambio de información procedural
    - Son lenguajes sencillos y eficientes
  - Declarativo
    - Intercambio de información declarativa
    - Lenguajes complejos
    - Problema de expresividad

Lenguajes de comunicación

- La comunicación entre agentes inteligentes está basada en el speech act theory
- Los agentes usan un conjunto de performativas pre-definidas para comunicar sus intenciones
- La semántica de las performativas permite al receptor interpretar el mensaje de forma adecuada
- Existen dos conjuntos de performativas utilizadas en los sistemas multi-agente
  - KQML: Knowledge Query and Manipulation Language
  - FIPA-ACL: Agent comunication language

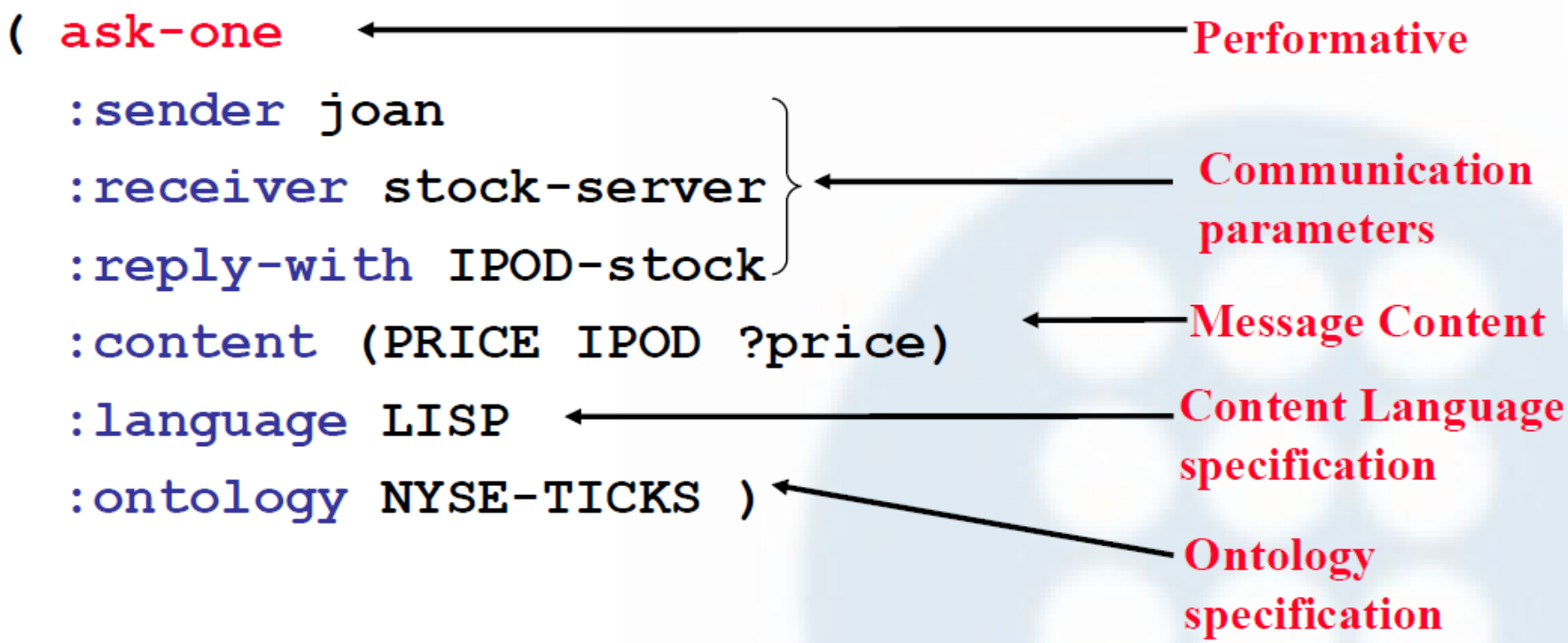
KQML

- Desarrollado por el *ARPA Knowledge Sharing Initiative*
- Compuesto de dos partes:
  - KQML: Knowledge Query and Manipulation Language
  - KIF: Content language

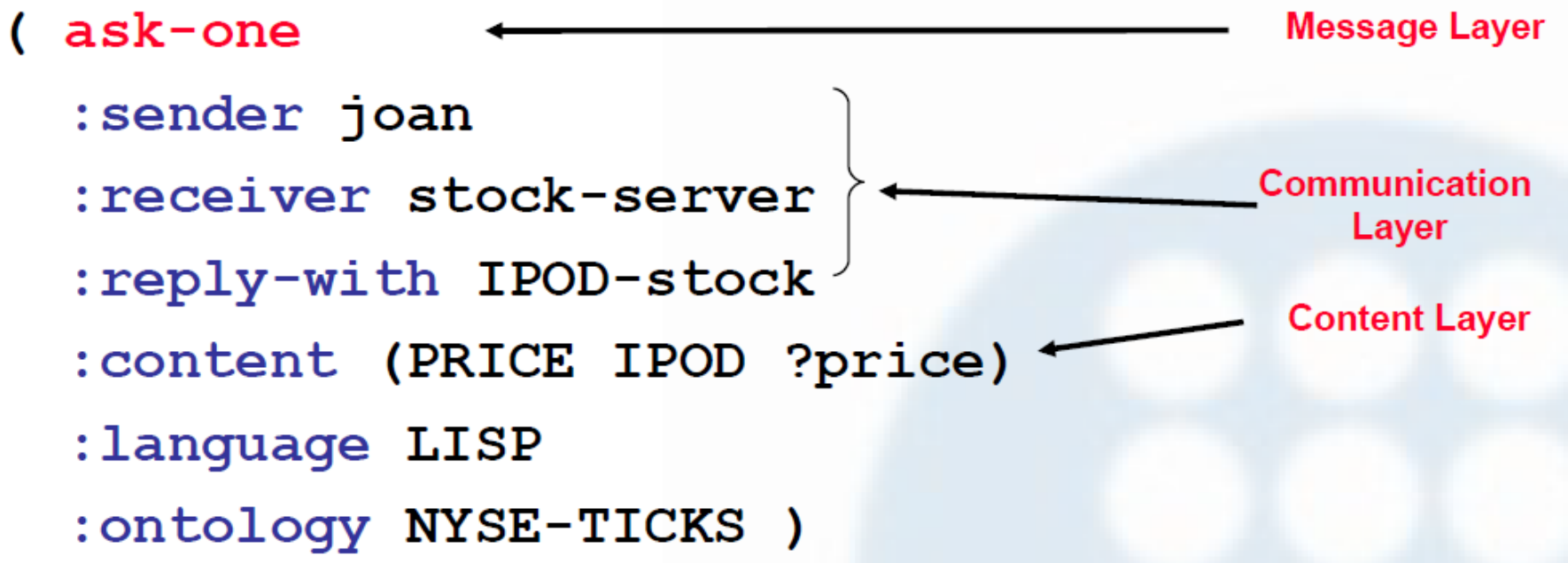
KQML

- Define un conjunto amplio de verbos comunicativos o performativas:
  - Basic requests (evaluate, ask-one, perform)
  - Multiagent requests (stream-in)
  - Responses (reply, sorry)
  - Information (tell, achieve, cancel)
  - Coordination (stand-by, ready, next)
  - Capabilities (advertise, subscribe)
  - Networking (register, forward, broadcast)

KQML


(ask-one
:sender joan
:receiver stock-server
:reply-with IPOD-stock
:content (PRICE IPOD ?price)
:language LISP
:ontology NYSE-TICKS)
Performative
Communication parameters
Message Content
Content Language specification
Ontology specification

KQML


(ask-one
:sender joan
:receiver stock-server
:reply-with IPOD-stock
:content (PRICE IPOD ?price)
:language LISP
:ontology NYSE-TICKS)
Message Layer
Communication Layer
Content Layer

FIPA-ACL

- Desarrollado por la FIPA (Foundation for Intelligent Physical Agents)
- Usado en JADE y otras plataformas
- Evolución de KQML
  - 22 performativas disponibles (reducción de KQML)
  - Comunicación
    - Sender, receiver
  - Content
    - Language, encoding, ontology
  - Conversation
    - Protocol, conversation-in, in-reply-to, etc.

FIPA-ACL

```
(inform
    :sender agent1
    :receiver  agent5
    :content(price good200 150)
    :language  sl
    :ontology  hpl-auction
)
```

FIPA-ACL

performative	passing info	requesting info	negotiation	performing actions	error handling
accept-proposal			x		
agree				x	
cancel		x		x	
cfp			x		
confirm	x				
disconfirm	x				
failure					x
inform	x				
inform-if	x				
inform-ref	x				
not-understood					x
propose			x		
query-if		x			
query-ref		x			
refuse				x	
reject-proposal			x		
request				x	
request-when				x	
request-whenever				x	
subscribe		x			

Protocolo de interacción

- Las performativas no son útiles solas, pero lo son como parte de la especificación de un protocolo
- Un protocolo es un diálogo entre agentes que sigue unas reglas que definen que performativas usar y cuando para poder alcanzar un objetivo específico
  - Simplifican el proceso de comunicación
  - El agente sabe que performativas puede usar o espera recibir en cada estado del protocolo
- Definen las secuencias de mensajes de un diálogo particular como una máquina de estados finitos determinista
- Cada protocolo está diseñado para un tipo de diálogo. Escoged sabiamente

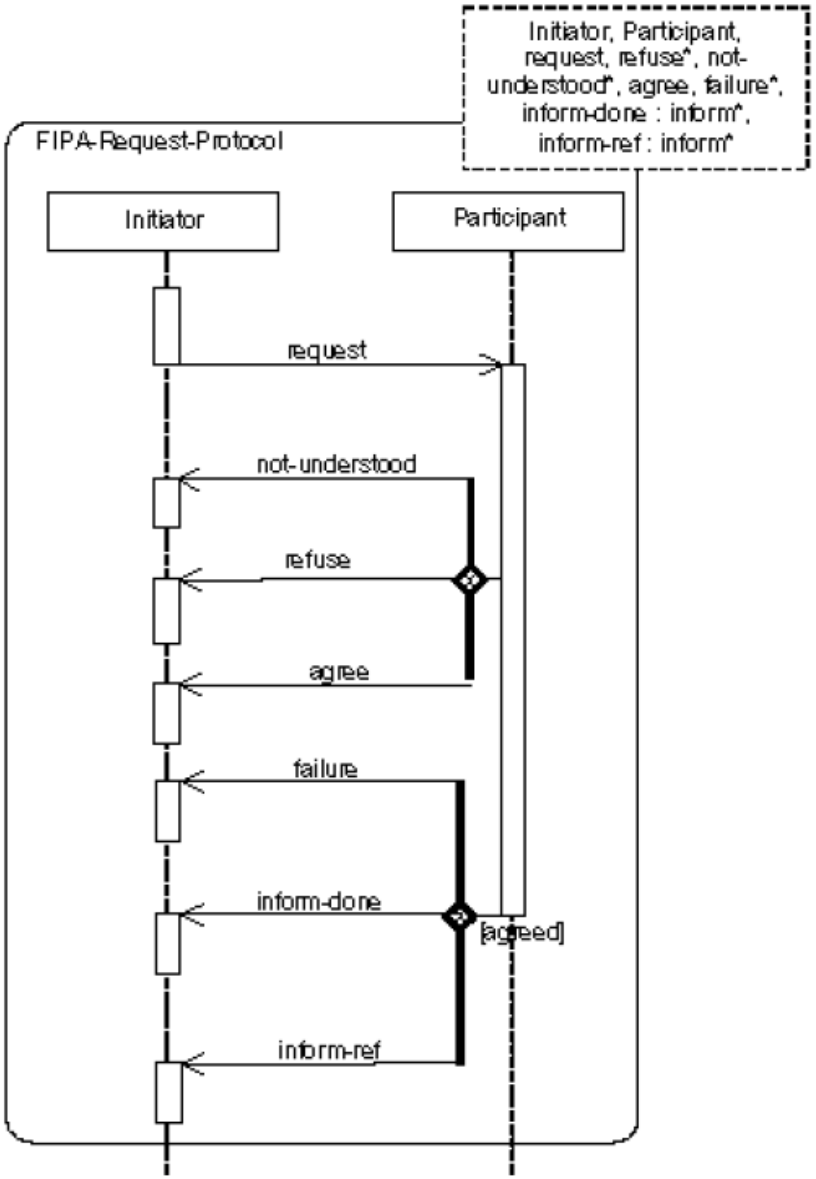
Protocolos FIPA

- Los más usados son:
  - Request: Pide a otro agente que ejecute una acción. El receptor devuelve los resultados de la ejecución, si es posible
  - Request-when: Pide a otro agente que ejecute una acción cuando se cumplan ciertas condiciones
  - Query: Pregunta a otro agente si una expresión determinada es cierta. Agentes encargados de la ontología

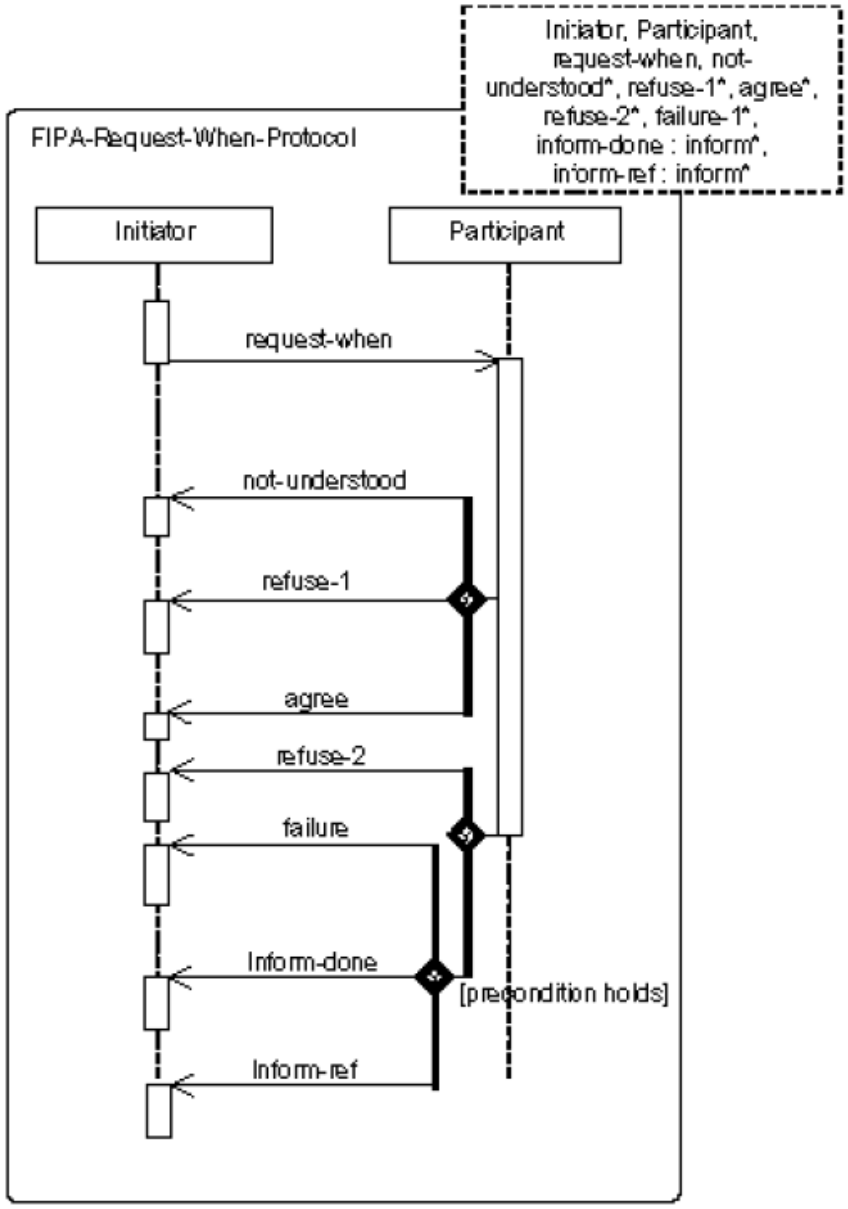
Protocolos FIPA

- Los más usados son:
  - Propose: Propone a otro agente que ejecute una acción bajo unas condiciones. El receptor acepta o rechaza la propuesta
  - Contract net: Protocolo complejo que permite a un grupo de agentes negociar propuestas para llevar a cambo una acción. El iniciador del protocolo recibe propuestas y selecciona la que más le interese.

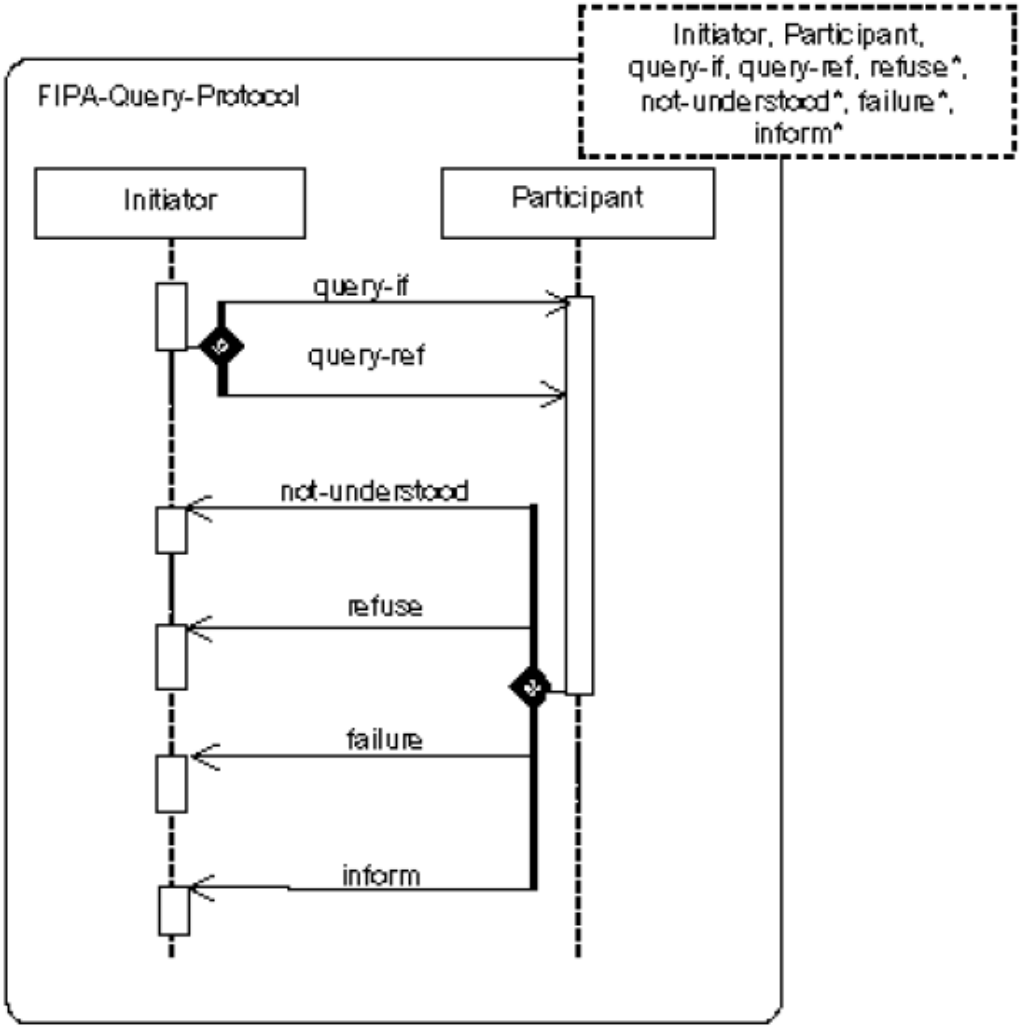
Protocolos FIPA



Protocolos FIPA

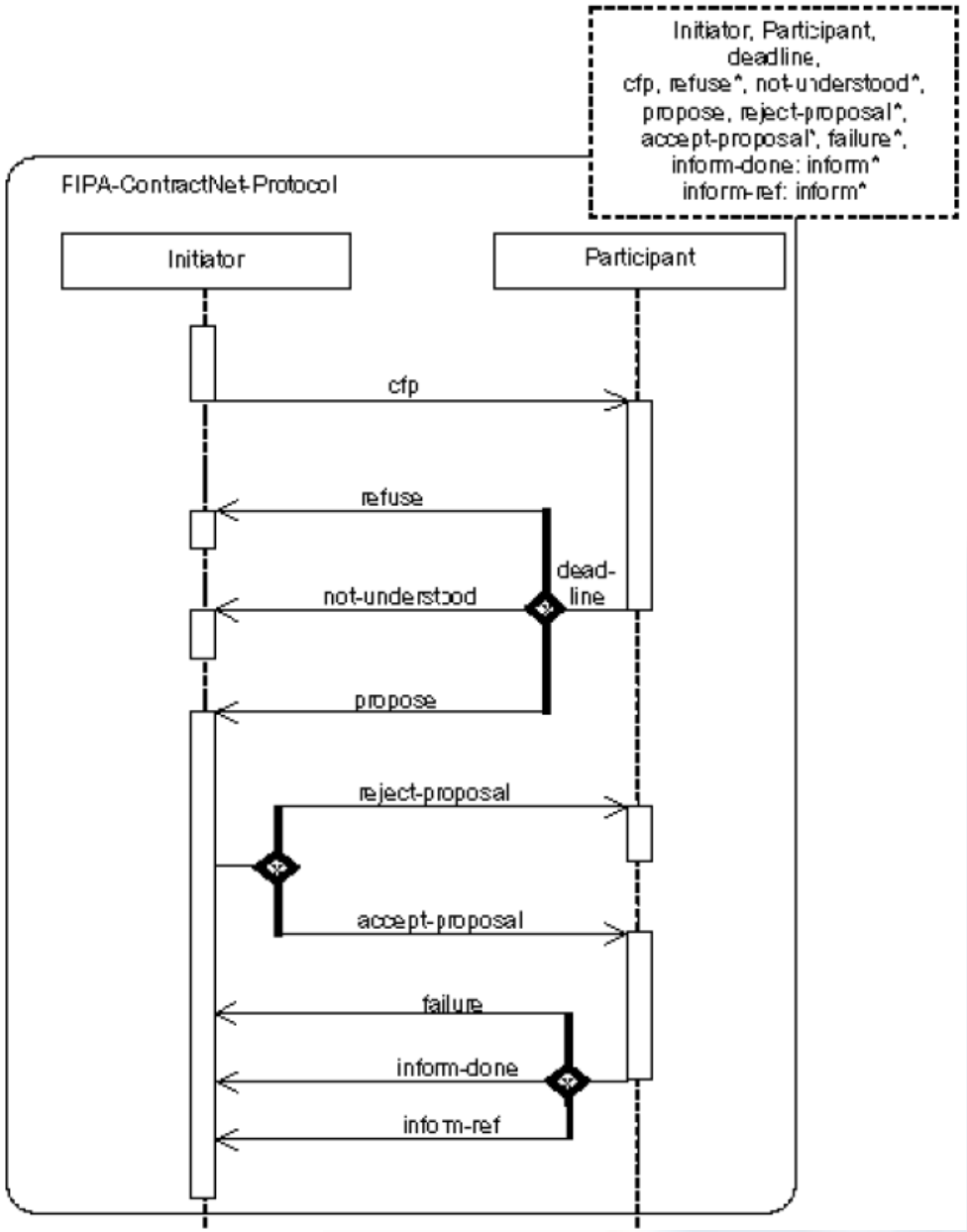


Protocolos FIPA

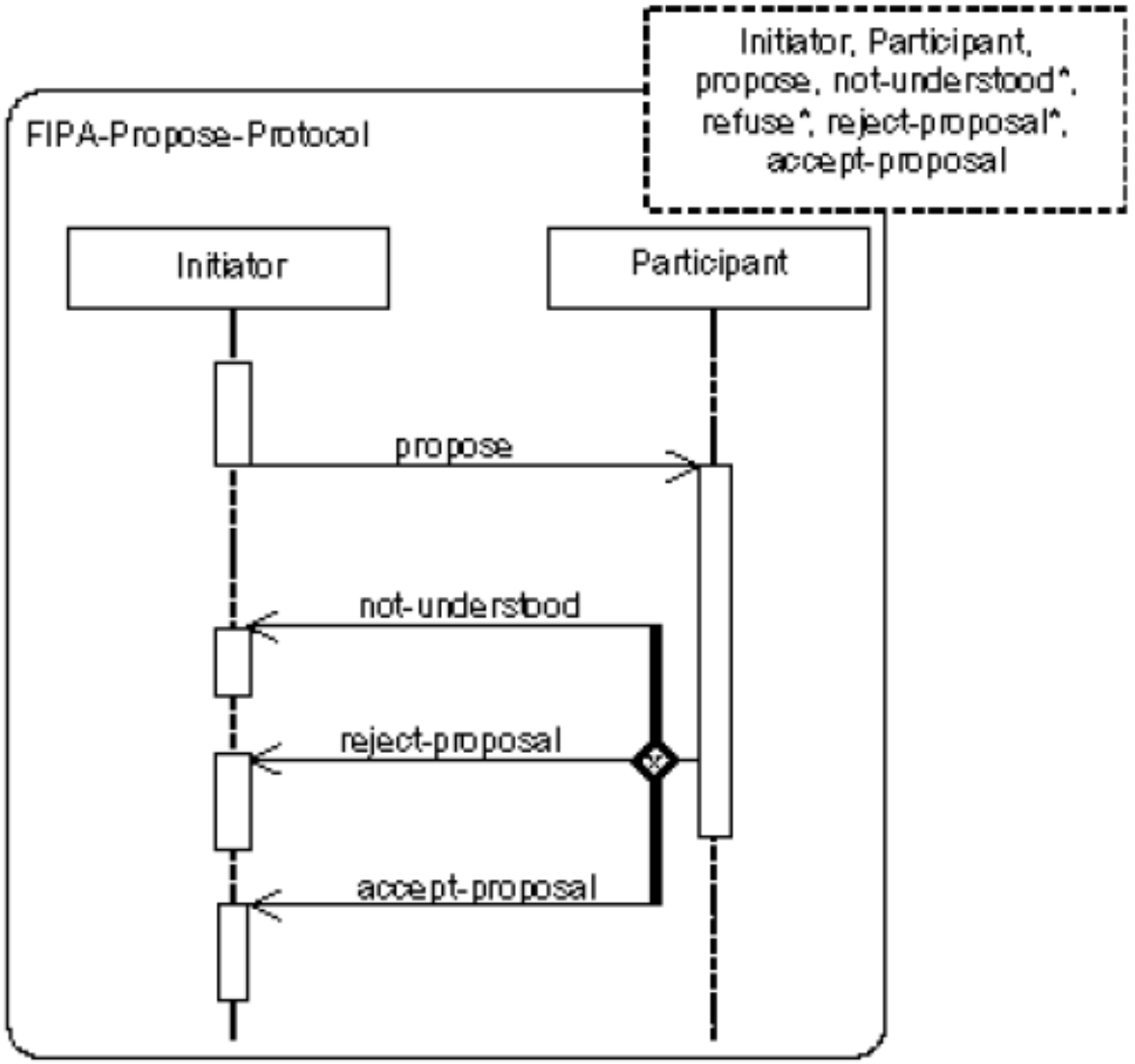


{ia, igomez}@lsi.upc.edu

Protocolos FIPA



Protocolos FIPA



Protocolos en Jade

```
ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setProtocol(FIPANames.InteractionProtocols.FIPA_REQUEST);
request.addReceiver(new AID("receiver", AID.ISLOCALNAME));
myAgent.addBehaviour( new AchieveREInitiator(myAgent, request) {
  protected void handleInform(ACLMessage inform) {
            System.out.println("Protocol finished. Rational Effect achieved.
Received the following message: "+inform);
      }
});
```

Protocolos en Jade

jade.proto

Class AchieveREResponder

java.lang.Object
- jade.core.behaviours.Behaviour
  - jade.core.behaviours.CompositeBehaviour
    - jade.core.behaviours.SerialBehaviour
      - jade.core.behaviours.FSMBehaviour
        - **jade.proto.AchieveREResponder**

```
MessageTemplate mt =

      AchieveREResponder.createMessageTemplate(FIPANames.InteractionProtocols.F
IPA_REQUEST);
myAgent.addBehaviour( new AchieveREResponder(myAgent, mt) {
  protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage
response) {
      System.out.println("Responder has received the following message: " +
request);
      ACLMessage informDone = request.createReply();
      informDone.setPerformative(ACLMessage.INFORM);
      informDone.setContent("inform done");
      return informDone;
      }
});
```

Agente: Comunicación I

- Los mensajes son instancias de jade.lang.acl.ACLMessage
  - **get/setPerformative();**
  - **get/setSender();**
  - **add/getAllReceiver();**
  - **get/setLanguage();**
  - **get/setOntology();**
  - **get/setContent();**

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

```
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);
```

Agente: Comunicación II

- El uso continuo de recieve() es ineficiente
- Uso de *block()*
- Bloquea el behaviour de la lista de behaviours del agente
  - No bloquea el agente
- Cada vez que llega un mensaje, se desbloquean los behaviours bloqueados y pueden leerlo

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

This is the strongly recommended pattern to receive messages within a behaviour

Agente: Comunicación III

- Es más seguro hacer uso de message templates

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");

public void action() {
  ACLMessage msg = myAgent.receive(tpl);
  if (msg != null) {
    // Process the message
  }
  else {
    block();
  }
}
```

Agente: Comunicación III

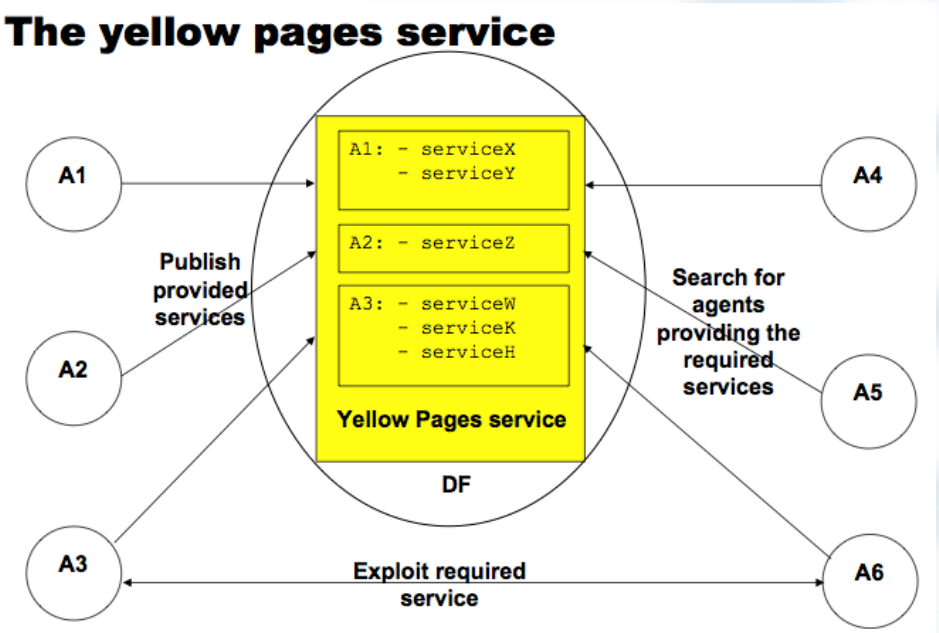
- Es más seguro hacer uso de message templates

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");

public void action() {
  ACLMessage msg = myAgent.receive(tpl);
  if (msg != null) {
    // Process the message
  }
  else {
    block();
  }
}
```

- Método *blockingReceive()* disponible
  - Pero peligroso

Agente: DF

- Clase jade.domain.DFService
  - Register()
  - Modify()
  - Deregister()
  - Search()



Tareas

- Agente Lover
  - Blocking behaviour
    - ¿Se bloquea el agente?
    - Recibir mensaje usando template
    - Enviar mensaje usando template
    - Uso de performativas
- Ejecutar el agente y comprobar su comportamiento
  - GUI
  - Consola
  - Usar el Sniffer en la GUI
- Entender el uso Uso del DF

Tareas

- Agentes con protocolos
  - ContractNetInitiatorAgent
  - ContractNetResponderAgent
- Ejecutar los agentes
  - Desde consola
  - Usar el Sniffer en la GUI para observarlos
  - Usar el introspector en la GUI para observarlos
  - Jugar con los parámetros del agente

```
--Complex agents based on protocols
java -cp /home/igomez/Jade/jade/lib/jade.jar:/home/igomez/NetBeansProjects/JadeApplication/dist/JadeApplication.jar jade.Boot
-local-host 127.0.0.1 -gui &
java -cp /home/igomez/Jade/jade/lib/jade.jar:/home/igomez/NetBeansProjects/JadeApplication/dist/JadeApplication.jar jade.Boot
-local-host 127.0.0.1 -container Timmy:org.upc.edu.Protocols.ContractNetResponderAgent\("YES","6"\) &
java -cp /home/igomez/Jade/jade/lib/jade.jar:/home/igomez/NetBeansProjects/JadeApplication/dist/JadeApplication.jar jade.Boot
-local-host 127.0.0.1 -container Jimmy:org.upc.edu.Protocols.ContractNetResponderAgent\("YES","4"\) &
java -cp /home/igomez/Jade/jade/lib/jade.jar:/home/igomez/NetBeansProjects/JadeApplication/dist/JadeApplication.jar jade.Boot
-local-host 127.0.0.1 -container Winnie:org.upc.edu.Protocols.ContractNetResponderAgent\("NO","0"\) &
java -cp /home/igomez/Jade/jade/lib/jade.jar:/home/igomez/NetBeansProjects/JadeApplication/dist/JadeApplication.jar jade.Boot
-local-host 127.0.0.1 -container Sesilu:org.upc.edu.Protocols.ContractNetInitiatorAgent\("Timmy","Jimmy","Winnie"\)
```