

# JADE PROGRAMMER'S GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 18-September-2000. JADE 2.0

Authors: Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (CSELT S.p.A.)  
Giovanni Rimassa (University of Parma)

Copyright (C) 2000 CSELT S.p.A.

JADE - Java Agent DEvelopment Framework is a framework to develop multi-agent systems in compliance with the FIPA specifications.

Copyright (C) 2000 CSELT S.p.A.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, version 2.1 of the License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

## TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION</b>	<b>5</b>
<b>2</b>	<b>JADE FEATURES</b>	<b>7</b>
<b>3</b>	<b>CREATING MULTI-AGENT SYSTEMS WITH JADE</b>	<b>7</b>
<b>3.1</b>	<b>The Agent Platform</b>	<b>7</b>
3.1.1	FIPA-Agent-Management ontology	9
3.1.1.1	Basic concepts of the ontology	10
3.1.2	Simplified API to access DF and AMS services	10
3.1.2.1	DFServiceCommunicator	10
3.1.2.2	AMSServiceCommunicator	11
<b>3.2</b>	<b>The Agent class</b>	<b>11</b>
3.2.1	Agent life cycle	12
3.2.1.1	Starting the agent execution	13
3.2.1.2	Stopping agent execution	13
3.2.2	Inter-agent communication.	14
3.2.2.1	Accessing the private queue of messages.	14
<b>3.3</b>	<b>Agent Communication Language (ACL) Messages</b>	<b>14</b>
3.3.1	Support to reply to a message	15
3.3.2	Support for Java serialisation and transmission of a sequence of bytes	15
3.3.3	The ACL Codec	15
3.3.4	The MessageTemplate class	16
<b>3.4</b>	<b>The agent tasks. Implementing Agent behaviours</b>	<b>16</b>
3.4.1	class Behaviour	19
3.4.2	class SimpleBehaviour	20
3.4.3	class OneShotBehaviour	20
3.4.4	class CyclicBehaviour	20
3.4.5	class ComplexBehaviour	20
3.4.6	class SequentialBehaviour	20
3.4.7	class NonDeterministicBehaviour	21
3.4.8	class SenderBehaviour	21
3.4.9	class ReceiverBehaviour	21
3.4.10	Examples	21
<b>3.5</b>	<b>Interaction Protocols</b>	<b>25</b>
3.5.1	FIPA-Request	26
3.5.1.1	FipaRequestInitiatorBehaviour	26
3.5.1.2	FipaRequestResponderBehaviour	26
3.5.2	FIPA-Query	27
3.5.2.1	FipaQueryInitiatorBehaviour	27
3.5.2.2	FipaQueryResponderBehaviour	28
3.5.3	FIPA-Contract-Net	28
3.5.3.1	FipaContractNetInitiatorBehaviour	29
3.5.4	FipaContractNetResponderBehaviour	30
<b>3.6</b>	<b>Application-defined content languages and ontologies</b>	<b>30</b>

3.6.1 Rationale	30
3.6.2 The conversion pipeline	31
3.6.3 Codec of a Content Language	32
3.6.4 Creating an Ontology	33
3.6.5 Application specific classes representing ontological roles	37
3.6.6 Discovering the ontological role of a Java object representing an entity in the domain of discourse	37
3.6.7 Setting and getting the content of an ACL message.	38
<b>3.7 Support for Agent Mobility</b>	<b>38</b>
3.7.1 JADE API for agent mobility.	39
3.7.2 JADE Mobility Ontology.	39
3.7.3 Accessing the AMS for agent mobility.	41
<b>4 A SAMPLE AGENT SYSTEM</b>	<b>44</b>
<b>5 RUNNING THE AGENT PLATFORM</b>	<b>44</b>
5.1 Software requirements	44
5.2 Getting the software	44
<b>5.3 Running JADE from the binary distribution</b>	<b>45</b>
5.3.1 Options available from the command line	45
5.3.2 Launching agents from the command line	45
5.3.3 Example	46
<b>5.4 Building JADE from the source distribution</b>	<b>46</b>
5.4.1 Building the JADE framework	46
5.4.2 Building JADE libraries	47
5.4.3 Building JADE HTML documentation	47
5.4.4 Building JADE examples and demo application	47
5.4.5 Cleaning up the source tree	48
<b>5.5 IIOP support and inter-platform messaging</b>	<b>48</b>
<b>6 GRAPHICAL USER INTERFACE TO MANAGE AND MONITOR THE AP ACTIVITY</b>	<b>48</b>
6.1 Remote Monitoring Agent	49
6.2 DummyAgent	50
6.3 DF GUI	51
6.4 Sniffer Agent	52
<b>7 RELEASE NOTES</b>	<b>53</b>
7.1 Major changes in JADE 2.01	53

<b>7.2 Major changes in JADE 2.0</b>	<b>53</b>	
7.2.1 Major changes		53
7.2.2 Guide to fast upgrade of the user code to JADE 2.0 and FIPA2000 specifications		55
<b>7.3 Major changes in JADE 1.4</b>	<b>58</b>	
<b>7.4 Major changes in JADE 1.3</b>	<b>58</b>	
<b>7.5 Major changes in JADE 1.2</b>	<b>59</b>	
<b>7.6 Major changes in JADE 1.1</b>	<b>59</b>	
<b>7.7 Major changes in JADE 1.0</b>	<b>59</b>	
<b>7.8 Major changes in Jade 0.97</b>	<b>60</b>	
<b>7.9 Major changes in Jade 0.92</b>	<b>60</b>	
<b>7.10 Major changes in Jade 0.9</b>	<b>61</b>	
<b>7.11 Major changes from JADE 0.71+ to JADE 0.79+</b>		<b>61</b>

---

## 1 INTRODUCTION

---

This programmer's guide is complemented by the HTML documentation available in the directory `jade/doc`. If and where conflict arises between what is reported in the HTML documentation and this guide, preference should be given to the HTML documentation that is updated more frequently.

JADE (Java Agent Development Framework) is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been fully coded in Java and an agent programmer, in order to exploit the framework, should code his/her agents in Java, following the implementation guidelines described in this programmer's guide.

This guide supposes the reader to be familiar with the FIPA standards<sup>1</sup>, at least with the *Agent Management* specifications (FIPA no. 23), the *Agent Communication Language*, and the *ACL Message Structure* (FIPA no. 61).

JADE is written in Java language and is made of various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Java was the programming language of choice because of its many attractive features, particularly geared towards object-oriented programming in distributed heterogeneous environments; some of these features are Object Serialization, Reflection API and Remote Method Invocation (RMI).

JADE is composed of the following main packages.

`jade.core` implements the kernel of the system. It owns the `Agent` class that must be extended by application programmers; besides, a `Behaviour` class hierarchy is contained in `jade.core.behaviours` sub-package. Behaviours implement the tasks, or intentions, of an agent. They are logical activity units that can be composed in various ways to achieve complex execution patterns and that can be concurrently executed. Application programmers define agent operations writing behaviours and agent execution paths interconnecting them.

The `jade.lang` package has a sub-package for every language used in JADE. In particular, a `jade.lang.acl` sub-package is provided to process Agent Communication Language according to FIPA standard specifications. `jade.lang.sl` contains the SL-0 codec<sup>2</sup>, both the parser and the encoder.

The `jade.onto` package contains a set of classes to support user-defined ontologies. It has a subpackage `jade.onto.basic` containing a set of basic concepts (i.e. `Action`, `TruePredicate`, `FalsePredicate`, ...) that are usually part of every ontology, and a `BasicOntology` that can be joined with user-defined ontologies.

The `jade.domain` package contains all those Java classes that represent the Agent Management entities defined by the FIPA standard, in particular the AMS and DF agents, that provide life-cycle, white and yellow page services. The subpackage `jade.domain.FIPAAgentManagement` contains the FIPA-Agent-Management Ontology and all the classes representing its concepts. The subpackage

---

<sup>1</sup> See <http://www.fipa.org/>

<sup>2</sup> refer to FIPA document no. 8 for the specifications of the SL content language.

`jade.domain.JADEAgentManagement` contains, instead, the JADE extensions for Agent-Management (e.g. for sniffing messages, controlling the life-cycle of agents, ...), including the Ontology and all the classes representing its concepts.

The `jade.gui` package contains a set of generic classes useful to create GUIs to display and edit Agent-Identifiers, Agent Descriptions, ACLMessages, ...

The `jade.mtp` package contains a Java interface that every Message Transport Protocol should implement in order to be readily integrated with the JADE framework, and the implementation of a set of these protocols.

`jade.proto` is the package that contains classes to model standard interaction protocols (i.e. *fipa-request*, *fipa-query*, *fipa-contract-net* and soon others defined by FIPA), as well as classes to help application programmers to create protocols of their own.

Finally, the `fipa` package contains the IDL module defined by FIPA for IIOP-based message transport.

JADE comes bundled with some tools that simplify platform administration and application development. Each tool is contained in a separate sub-package of `jade.tools`. Currently, the following tools are available:

- *Remote Management Agent, RMA* for short, acting as a graphical console for platform management and control. A first instance of an RMA can be started with a command line option ("*-gui*") , but then more than one GUI can be activated. JADE maintains coherence among multiple RMAs by simply multicasting events to all of them. Moreover, the RMA console is able to start other JADE tools.
- The *Dummy Agent* is a monitoring and debugging tool, made of a graphical user interface and an underlying JADE agent. Using the GUI it is possible to compose ACL messages and send them to other agents; it is also possible to display the list of all the ACL messages sent or received, completed with timestamp information in order to allow agent conversation recording and rehearsal.
- The *Sniffer* is an agent that can intercept ACL messages while they are in flight, and displays them graphically using a notation similar to UML sequence diagrams. It is useful for debugging your agent societies by observing how they exchange ACL messages.
- The *SocketProxyAgent* is a simple agent, acting as a bidirectional gateway between a JADE platform and an ordinary TCP/IP connection. ACL messages, travelling over JADE proprietary transport service, are converted to simple ASCII strings and sent over a socket connection. Viceversa, ACL messages can be tunnelled via this TCP/IP connection into the JADE platform. This agent is useful, e.g. to handle network firewalls or to provide platform interactions with Java applets within a web browser.
- The *DF GUI* is a complete graphical user interface that is used by the default Directory Facilitator (DF) of JADE and that can also be used by every other DF that the user might need. In such a way, the user might create a complex network of domains and sub-domains of yellow pages. This GUI allows in a simple and intuitive way to control the knowledge base of a DF, to federate a DF with other DF's, and to remotely control (register/deregister/modify/search) the knowledge base of the parent DF's and also the children DF's (implementing the network of domains and sub-domains).

JADE™ is a trade mark registered by CSELT.

---

## 2 JADE FEATURES

---

The following is the list of features that JADE offers to the agent programmer:

- Distributed agent platform. The agent platform can be split among several hosts (provided they can be connected via RMI). Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as Java threads and live within *Agent Containers* that provide the runtime support to the agent execution.
- Graphical user interface to manage several agents and agent containers from a remote host.
- Debugging tools to help in developing multi agents applications based on JADE.
- Intra-platform agent mobility, including state and code of the agent.
- Support to the execution of multiple, parallel and concurrent agent activities via the behaviour model. JADE schedules the agent behaviours in a non-preemptive fashion.
- FIPA-compliant Agent Platform, which includes the *AMS (Agent Management System)*, the *DF (Directory Facilitator)*, and the *ACC (Agent Communication Channel)*. All these three components are automatically activated at the agent platform start-up.
- FIPA-compliant IIOP-based message transport system to connect different agent platforms.
- Many FIPA-compliant DFs can be started at run time in order to implement multi-domain applications, where a domain is a logical set of agents, whose services are advertised through a common facilitator. Each DF inherits a GUI and all the standard capabilities defined by FIPA (i.e. capability of registering, deregistering, modifying and searching for agent descriptions; and capability of federating within a network of DF's).
- Efficient transport of ACL messages inside the same agent platform. Infact, messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures. When crossing platform boundaries, the message is automatically converted to/from the FIPA compliant syntax, encoding, and transport protocol. This conversion is transparent to the agent implementers that only need to deal with Java objects.
- Library of FIPA interaction protocols ready to be used.
- Automatic registration and deregistration of agents with the AMS.
- FIPA-compliant naming service: at start-up agents obtain their GUID (Globally Unique Identifier) from the platform.
- Support to the usage of application-defined content languages and ontologies.

---

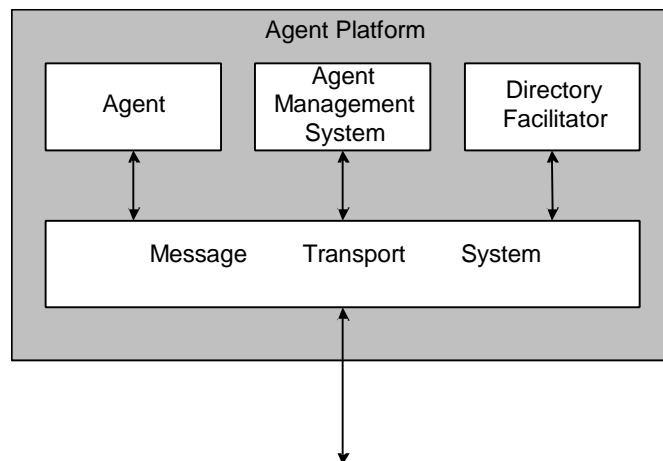
## 3 CREATING MULTI-AGENT SYSTEMS WITH JADE

---

This chapter describes the JADE classes that support the development of multi-agent systems. JADE warrants syntactical compliance and, where possible, semantic compliance with FIPA specifications.

### 3.1 The Agent Platform

The standard model of an agent platform, as defined by FIPA, is represented in the following figure.



*Figure 1 - Reference architecture of a FIPA Agent Platform*

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

JADE fully complies with this reference architecture and when a JADE platform is launched, the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provides a complete run-time environment for the execution of any set of JADE agents.



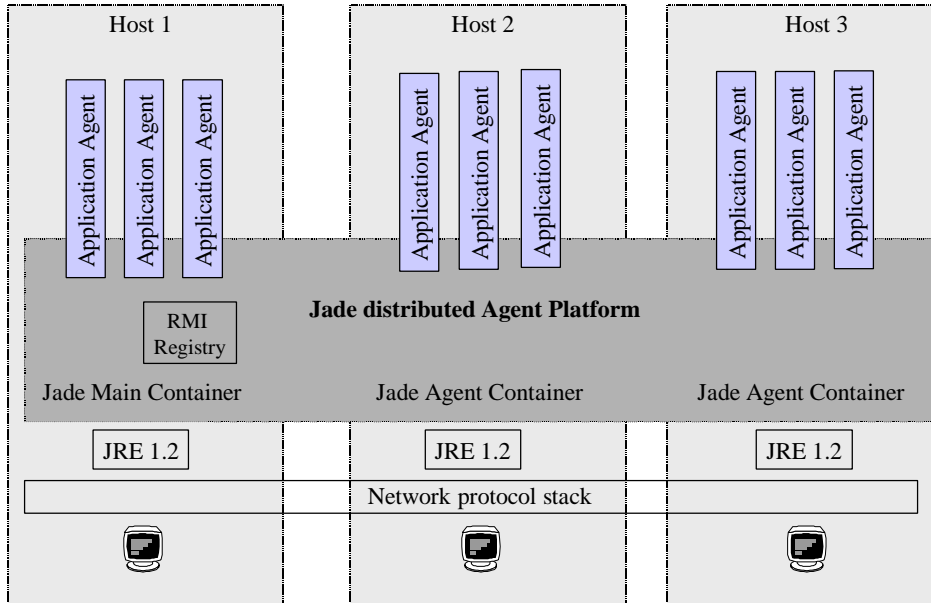


Figure 2 - JADE Agent Platform distributed over several containers

According to the FIPA specifications, DF and AMS agents communicate by using the FIPA-SL0 content language, the `fipa-agent-management` ontology, and the `fipa-request` interaction protocol. JADE provides compliant implementations for all these components:

- the SL-0 content language is implemented by the class `jade.lang.sl.SL0Codec`. Automatic capability of using this language can be added to any agent by using the method `Agent.registerLanguage(SL0Codec.NAME, new SL0Codec());`
- all the concepts of the ontology (apart from the Agent Identifier that is implemented by `jade.core.AID`) are implemented by the classes in the `jade.domain.FIPAAgentManagement` package. The class `FIPAAgentManagementOntology` defines the vocabulary with all the constant symbols of the ontology. Automatic capability of using this ontology can be added to any agent by using the method `Agent.registerOntology(FIPAAgentManagementOntology.NAME, FIPAAgentManagementOntology.instance());`
- finally, the `fipa-request` interaction protocol is implemented as ready-to-use behaviours in the package `jade.proto`.

### 3.1.1 FIPA-Agent-Management ontology

Every class implementing a concept of the `fipa-agent-management` ontology is a simple collection of attributes, with public methods to read and write them, according to the frame based model that represents FIPA `fipa-agent-management` ontology concepts. The following convention has been used. For each attribute of the class, named `attrName` and of type `attrType`, two cases are possible:

- 1) The attribute type is a single value; then it can be read with `attrType getAttrName()` and written with `void setAttrName(attrType a)`, where every call to `setAttrName()` overwrites any previous value of the attribute.
- 2) The attribute type is a set or a sequence of values; then there is an `void addAttrName(attrType a)` method to insert a new value and a `void clearAllAttrName()` method to remove all the values (the list becomes empty). Reading is performed by a `Iterator` `getAllAttrName()` method that returns an `Iterator` that allows the programmer to walk through the `List` and cast its elements to the appropriate type.

Refer to the javadoc for a complete list of these classes and their interface.

### 3.1.1.1 Basic concepts of the ontology

The package `jade.onto.basic` includes a set of classes that are commonly part of every ontology, such as `Action`, `TruePredicate`, `FalsePredicate`, `ResultPredicate`, ... The `BasicOntology` can be joined to any user-defined ontology as described in section 3.6.

Notice that the `Action` class should be used to represent actions. It has a couple of methods to set/get the AID of the actor (i.e. the agent who should perform the action) and the action itself (e.g. `Register/Deregister/Modify`).

### 3.1.2 Simplified API to access DF and AMS services

JADE features described so far allow complete interactions between FIPA system agents and user defined agents, simply by sending and receiving messages as defined by the standard.

However, because those interactions have been fully standardized and because they are very common, the following classes allow to successfully accomplish this task with a simplified interface.

Two methods are implemented by the class `Agent` to get the AID of the default DF and AMS of the platform: `getDefaultDF()` and `getAMS()`.

#### 3.1.2.1 DFServiceCommunicator

`jade.domain.DFServiceCommunicator` implements a set of static methods to communicate with a standard FIPA DF service (i.e. yellow page).

It includes methods to `register`, `deregister`, `modify` and `search` with a DF. Each of this method has version with all the needed parameters, or with a subset of them where, those parameters that can be omitted have been defaulted to the default DF of the platform, the AID of the sending agent, the default Search Constraints.

Notice that all these methods blocks every activity of the agent until the action (i.e. `register/deregister/modify/search`) has been successfully executed or a `jade.domain.FIPAException` exception has been thrown (e.g. because a `FAILURE` message has been received by the DF).

In some cases, instead, it is more convenient to execute this task in a non-blocking way. The method `getNonBlockingBehaviour()` returns a non-blocking behaviour of type `RequestFIPASERVICEBehaviour` that can be added to the queue of the agent behaviours,

as usual, by using `Agent.addBehaviour()`. Several ways are available to get the result of this behaviour and the programmer can select one according to his preferred programming style:

- call `getLastMsg()` and `getSearchResults()` where both throw a `NotYetReadyException` if the task has not yet finished;
- create a `SequentialBehaviour` composed of two sub-behaviours: the first subbehaviour is the returned `RequestFIPAServiceBehaviour`, while the second one is application-dependent and is executed only when the first is terminated;
- use directly the class `RequestFIPAServiceBehaviour` by extending it and overriding all the `handleXXX` methods that handle the states of the fipa-request interaction protocol.

### 3.1.2.2 *AMSServiceCommunicator*

This class is the dual of `DFServiceCommunicator` to access the service provided by a standard FIPA AMS agent and it has exactly the same interface.

Notice that JADE calls automatically the `register` and `deregister` methods with the default AMS respectively before calling `setup()` method and just after `takeDown()` method returns; so there is no need for a normal programmer to call them.

However, under certain circumstances, a programmer might need to call its methods. To give some examples: when an agent wishes to register with the AMS of a remote agent platform, or when an agent wishes to modify its description by adding a private address to the set of its addresses, ...

## 3.2 The Agent class

The `Agent` class represents a common base class for user defined agents. Therefore, from the point of view of the programmer, a JADE agent is simply an instance of a user defined Java class that extends the base `Agent` class. This implies the inheritance of features to accomplish basic interactions with the agent platform (such as registration, configuration, remote management, ...) and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, ...).

The assumed computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours (refer to section 3.2 for implementation of behaviours). A scheduler, internal to the base `Agent` class and hidden to the programmer, automatically manages the scheduling of behaviours.

### 3.2.1 Agent life cycle

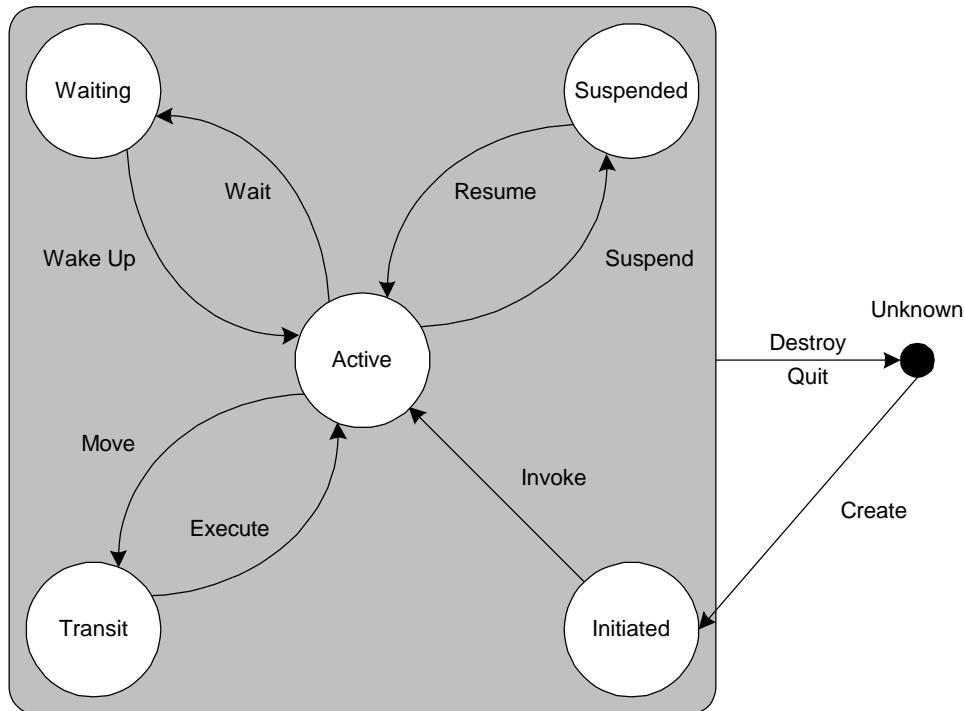


Figure 3 - Agent life-cycle as defined by FIPA.

A JADE agent can be in one of several states, according to Agent Platform Life Cycle in FIPA specification; these are represented by some constants in `Agent` class. The states are:

- **AP\_INITIATED** : the Agent object is built, but hasn't registered itself yet with the AMS, has neither a name nor an address and cannot communicate with other agents.
- **AP\_ACTIVE** : the Agent object is registered with the AMS, has a regular name and address and can access all the various JADE features.
- **AP\_SUSPENDED** : the Agent objects is currently stopped. Its internal thread is suspended and no agent behaviour is being executed.
- **AP\_WAITING** : the Agent object is blocked, waiting for something. Its internal thread is sleeping on a Java monitor and will wake up when some condition is met (typically when a message arrives).
- **AP\_DELETED** : the Agent is definitely dead. The internal thread has terminated its execution and the Agent is no more registered with the AMS.
- **AP\_TRANSIT** : a mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will be then sent to its new location.
- **AP\_COPY** : this state is internally used by JADE for agent being cloned.
- **AP\_GONE** : this state is internally used by JADE when a mobile agent has migrated to a new location and has a stable state.

The `Agent` class provides public methods to perform transitions between the various states; these methods take their names from a suitable transition in the Finite State Machine shown in FIPA specification *Agent Management*. For example, `doWait()` method puts the agent into `AP_WAITING` state from `AP_ACTIVE` state, `doSuspend()` method puts the agent into

AP\_SUSPENDED state from AP\_ACTIVE or AP\_WAITING state, ... Refer to the javadoc documentation of the Agent class for a complete list of these doXXX() methods.

Notice that an agent is allowed to execute its behaviours (i.e. its tasks) only when it is in the AP\_ACTIVE state. Take care that **if any behaviours call the `doWait()` method, then the whole agent and all its activities are blocked and not just the calling behaviour**. Instead, the `block()` method is part of the Behaviour class in order to allow suspending a single agent behaviour (see section 3.3 for details on the usage of behaviours).

### 3.2.1.1 Starting the agent execution

The JADE framework controls the birth of a new agent according to the following steps: the agent constructor is executed, the agent is given an identifier (see `jade.core.AID`), it is registered with the AMS, it is put in the AP\_ACTIVE state, and finally the `setup()` method is executed. According to the FIPA specifications, an agent identifier comprises the following attributes:

- a globally unique name. JADE composes this name as the concatenation of the local name – i.e. the agent name provided on the command line – plus the '@' symbol, plus the home agent platform identifier – i.e. `<hostname> ':' <port number of the JADE RMI registry> '/' 'JADE'`;
- a set of agent addresses. Each agent inherits the transport addresses of its home agent platform;
- a set of resolvers, i.e. white page services with which the agent is registered. JADE put, as a default, the AID of the AMS.

The `setup()` method is therefore the point where any application-defined agent activity starts. It is required in fact that the programmer implements the `setup()` method in order to insert the initialisation functions of the agent. When the `setup()` method is executed, the agent has been already registered with the AMS and its Agent Platform state is AP\_ACTIVE. The programmer should use this initialisation procedure to:

- (optional) if necessary, modify the data registered with the AMS (see section 3.4);
- (optional) set the description of the agent and its provided services and, if necessary, register the agent with one or more domains, i.e. DFs (see section 3.4);
- (necessary) add tasks to the queue of ready tasks, by using the method `addBehaviour()`. These behaviours are scheduled immediately after the end of the `setup()` method;

For a correct implementation, at least one behaviour should be added within the method `setup()`. At the end of the method `setup()`, JADE automatically executes the first behaviour in the queue of ready tasks and then switch to the other behaviours in the queue by using a round-robin non-preemptive scheduler. The two methods `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)` of the Agent class can be used to manage this queue of tasks.

### 3.2.1.2 Stopping agent execution

Any behaviour can call the Agent method `doDelete()` in order to stop the agent execution and kill the agent.

The `takeDown()` method should be overridden by the programmers in order to implement the cleaning up functions of the agent. It is in fact executed when the agent is about to go to AP\_DELETED state, i.e. it is going to be destroyed. When this method is executed the agent is

still registered with the AMS and can therefore send messages to other agents, but just after the `takeDown()` method is completed, the agent will be de-registered and its thread destroyed. The intended purpose of this method is to perform application specific cleanup operations, such as de-registering with DF agents.

### 3.2.2 Inter-agent communication.

The `Agent` class provides also a set of methods for inter-agent communication. According to the FIPA specification, agents communicate via asynchronous message passing, where objects of the `ACLMessage` class are the exchanged payloads. See also section 3.2 for a description of the `ACLMessage` class. Some of the interaction protocols defined by FIPA are also provided as ready-to-use behaviours that can be scheduled for agent activities; they are part of the `jade.proto` package.

The `send()` method allows to send an `ACLMessage`. The value of `ACLMessage.receiver` slot indicates the list of the receiving agent names, according to the FIPA specifications. The method call is completely transparent to where the agent resides, i.e. be it local or remote, it is the platform that takes care of selecting the most appropriate address and transport mechanism.

#### 3.2.2.1 Accessing the private queue of messages.

All the messages received by an agent are put in its private queue by the agent platform. Several access modes have been implemented in order to get messages from this private queue:

- first of all it can be accessed in a blocking (i.e. `blockingReceive()` method) or non-blocking way (i.e. `receive()` method). The blocking version must be used very carefully because it *causes the suspension of all the agent activities and in particular of all its Behaviours*. The non-blocking version returns immediately `null` when the requested message is not present in the queue;
- both methods can be augmented with a pattern-matching capability where a parameter is passed that describes the pattern of the requested `ACLMessage`. Section 3.2.3 describes the `MessageTemplate` class;
- the blocking-based access can be augmented with a timeout parameter. It is a *long* that describes the maximum number of milliseconds that the agent activity should remain blocked waiting for the requested message. If the timeout elapses before the message arrives, the method returns `null`;
- finally, the two behaviours `ReceiverBehaviour` and `SendBehaviour` can be used to schedule agent tasks that requires receiving or sending messages.

### 3.3 Agent Communication Language (ACL) Messages

The class `ACLMessage` represents ACL messages that can be exchanges between agents. It contains a set of attributes as defined by the FIPA specifications.

An agent willing to send a message should create a new `ACLMessage` object, fill its attributes with appropriate values, and finally call the method `send()` implemented by the class `Agent`). Likewise, an agent willing to receive a message should call `receive()` or `blockingReceive()` methods, both implemented by the `Agent` class and described in section 3.2.2.

Sending or receiving messages can also be scheduled as independent agent activities by adding the behaviours `ReceiverBehaviour` and `SendBehaviour` to the `Agent` queue of tasks.

All the attributes of the `ACLMessage` object can be accessed via the `set/get<Attribute>()` access methods. All attributes are named after the names of the parameters, as defined by the FIPA specifications. Those parameters whose type is a set of values (like `receiver`, for instance) can be accessed via the methods `add/getAll<Attribute>()` where the first method adds a value to the set, while the second method returns an `Iterator` over all the values in the set. Notice that all the `get` methods return `null` when the attribute has not been yet set.

Furthermore, this class also defines a set of constants that should be used to refer to the FIPA performatives, i.e. `REQUEST`, `INFORM`, etc. When creating a new ACL message object, one of these constants must be passed to `ACLMessage` class constructor, in order to select the message performative.

The `reset()` method resets the values of all message fields.

The `toString()` method returns a `String` representing the message. This method should be just used for debugging purposes.

### 3.3.1 Support to reply to a message

According to FIPA specifications, a reply message must be formed taking into account a set of well-formed rules, such as setting the appropriate value for the attribute *in-reply-to*, using the same *conversation-id*, etc. JADE helps the programmer in this task via the method `createReply()` of the `ACLMessage` class. This method returns a new `ACLMessage` object that is a valid reply to the current one. Then, the programmer only needs to set the application-specific communicative act and message content.

### 3.3.2 Support for Java serialisation and transmission of a sequence of bytes

Some applications may benefit from transmitting a sequence of bytes over the content of an `ACLMessage`. A typical usage is passing Java objects between two agents by exploiting the Java serialization. The `ACLMessage` class supports the programmer in this task by allowing the usage of *Base64* encoding through the two methods `setContentObject()` and `getContentObject()`. Refer to the HTML documentation of the JADE API and to the examples in `examples/Base64` directory for an example of usage of this feature.

It must be noticed that this feature does not comply to FIPA and that any agent platform can recognize automatically the usage of Base64 encoding<sup>3</sup>, so the methods must appropriately used by the programmers and should suppose that communicating agents know a-priori the usage of these methods.

### 3.3.3 The ACL Codec

Under normal conditions, agents never need to call explicitly the codec of the ACL messages because it is done automatically by the platform. However, when needed for some special circumstances, the programmer should use the methods provided by the class `StringACLCodec` to parse and encode ACL messages in `String` format.

---

<sup>3</sup> The implementation of this feature uses the source code contained within the `src/starlight` directory. This code is covered by the GNU General Public License, as decided by the copyright owner Kevin Kelley. The GPL license itself has been included as a text file named `COPYING` in the same directory. If the programmer does not need any support for Base64 encoding, then this code is not necessary and can be removed.

### 3.3.4 The MessageTemplate class

The JADE behaviour model allows an agent to execute several parallel tasks. However any agent should be provided with the capability of carrying on also many simultaneous conversations. Because the queue of incoming messages is shared by all the agent behaviours, an access mode to that queue based on pattern matching has been implemented (see 3.2.2.1).

The `MessageTemplate` class allows to build patterns to match ACL messages against. Using the methods of this class the programmer can create one pattern for each attribute of the `ACLMessage`. Elementary patterns can be combined with AND, OR and NOT operators, in order to build more complex matching rules.

In such a way, the queue of incoming `ACLMessages` can be accessed via pattern-matching rather than FIFO.

### 3.4 The agent tasks. Implementing Agent behaviours

An agent must be able to carry out several concurrent tasks in response to different external events. In order to make agent management efficient, every JADE agent is composed of a single execution thread and all its tasks must be implemented as `Behaviour` objects.

The developer who wants to implement an agent-specific task should define one or more `Behaviour` subclasses, instantiate them and add the behaviour objects to the agent task list. The `Agent` class, which must be extended by agent programmers, exposes two methods: `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)`, which allow to manage the ready tasks queue of a specific agent. Notice that behaviours and sub-behaviours can be added whenever is needed, and not only within `Agent.setup()` method. Adding a behaviour should be seen as a way to spawn a new (cooperative) execution thread within the agent.

A scheduler, implemented by the base `Agent` class and hidden to the programmer, carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue, executing a `Behaviour`-derived class until it will release control (this happens when `action()` method returns). If the task relinquishing the control has not yet completed, it will be rescheduled the next round. A behaviour can also block, waiting for a message to arrive. In detail, the agent scheduler executes `action()` method of each behaviour present in the ready behaviours queue; when `action()` returns, method `done()` is called to check if the behaviour has completed its task. If so, the behaviour object is removed from the queue.

Behaviours work just like co-operative threads, but there is no stack to be saved. ***Therefore, the whole computation state must be maintained in instance variables of the Behaviour and its associated Agent.***

In order to avoid an active wait for messages (and, as a consequence, a waste of CPU time), every single `Behaviour` is allowed to block its computation. ***Method `block()` puts the behaviour in a queue of blocked behaviours and takes effect as soon as `action()` returns.*** All blocked behaviours are rescheduled as soon as a new message arrives. Moreover, a behaviour object can block itself for a limited amount of time passing a timeout value to `block()` method. In future releases of JADE, more wake up events will be probably considered. The programmer must take care to block again a behaviour if it was not interested in the arrived message.

**Because of the non preemptive multitasking model chosen for agent behaviours, agent programmers must avoid to use endless loops and even to perform long operations within `action()` methods. Remember that when some behaviour's `action()` is running, no other behaviour can go on until the end of the method (of course this is true only with respect**



to behaviours of the same agent: behaviours of other agents run in different Java threads and can still proceed independently).

Besides, since no stack context is saved, every time `action()` method is run from the beginning: there is no way to interrupt a behaviour in the middle of its `action()`, yield the CPU to other behaviours and then start the original behaviour back from where it left.

For example, suppose a particular operation `op()` is too long to be run in a single step and is therefore broken in three sub-operations, named `op1()`, `op2()` and `op3()`. To achieve desired functionality one must call `op1()` the first time the behaviour is run, `op2()` the second time and `op3()` the third time, after which the behaviour must be marked as terminated. The code will look like the following:

```
public class my3StepBehaviour {
    private int state = 1;
    private boolean finished = false;

    public void action() {
        switch (state) {
            case 1: { op1(); state++; break; }
            case 2: { op2(); state++; break; }
            case 3: { op3(); state=1; finished = true; break; }
        }
    }

    public boolean done() {
        return finished;
    }
}
```

Following this idiom, agent behaviours can be described as finite state machines, keeping their whole state in their instance variables.

When dealing with complex agent behaviours (as agent interaction protocols) using explicit state variables can be cumbersome; so JADE also supports a compositional technique to build more complex behaviours out of simpler ones.

The framework provides ready to use Behaviour subclasses that can contain sub-behaviours and execute them according to some policy. For example, a `SequentialBehaviour` class is provided, that executes its sub-behaviours one after the other for each `action()` invocation.

The following figure is an annotated UML class diagram for JADE behaviours.

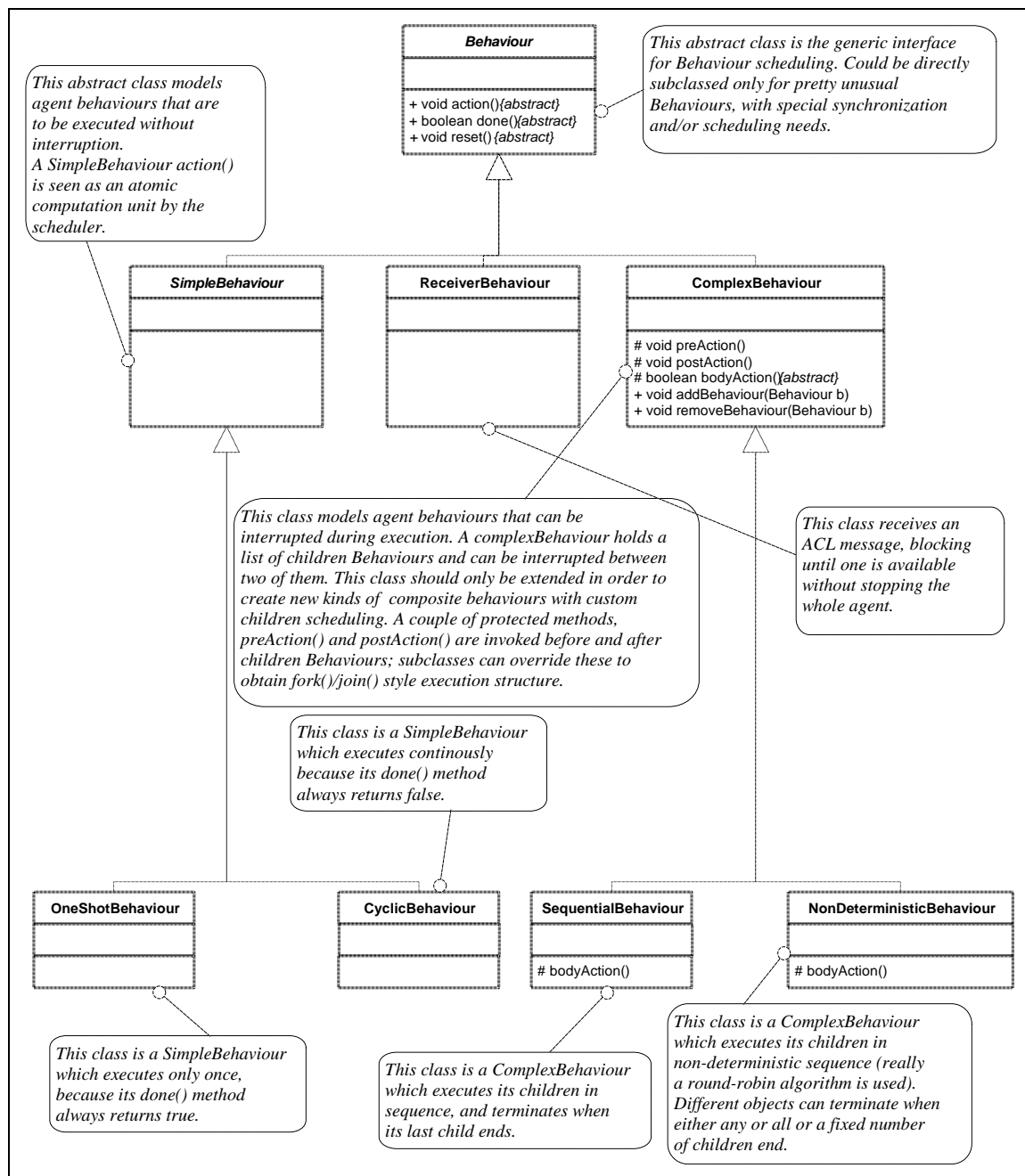


Figure D - UML Model of the Behaviour class hierarchy

Starting from the basic class Behaviour, a class hierarchy is defined in JADE framework.

Behaviour is an abstract class that provides the skeleton of the elementary task to be performed. It exposes various methods, two of which are most important during normal operation: the action() method, representing the actual task to be accomplished by the specific behaviour classes; and the done() method, used by the agent scheduler, that must return true when the behaviour is finished and can be removed from the queue, false when the behaviour has not yet finished and the action() method must be executed again.

The JADE class `SimpleBehaviour` can be used by the agent developer to implement atomic actions of the agent work.

`ComplexBehaviour` defines a method `addSubBehaviour(Behaviour)` and a method `removeSubBehaviour(Behaviour)`, allowing the agent writer to define complex behaviours made of several sub-behaviours. Since `ComplexBehaviour` extends `Behaviour`, the agent writer has the possibility to implement a structured tree composed of behaviours of different kinds (including `ComplexBehaviours` themselves). The agent scheduler only consider the top-most tasks for its scheduling policy: during each "time slice" (which, in practice, corresponds to one execution of the `action()` method) assigned to an agent task only a single subtask is executed. Each time a top-most task returns, the agent scheduler assigns the control to the next task in the ready queue.

`OneShotBehaviour` is an abstract class that models atomic behaviours that must be executed only once. Its `done()` method always returns `true`, so `action()` is executed exactly once.

`CyclicBehaviour` is an abstract class that models atomic behaviours that never end and must be executed for ever. Its `done()` method always returns `false`, so `action()` is executed whenever the current behaviour gets its time slice.

`SequentialBehaviour` is a `ComplexBehaviour` that executes its sub-behaviours sequentially, blocks when its current child is blocked and terminates when all its sub-behaviours are done.

`NonDeterministicBehaviour` is a `ComplexBehaviour` that executes its sub-behaviours non deterministically, blocks when all its children are blocked and terminates when a certain condition on its sub-behaviours is met. The following alternative conditions have been implemented: ending when all its sub-behaviours are done, when any sub-behaviour terminates or when `N` sub-behaviours have finished.

Two more classes are supplied which carry out specific action of general utility: `SenderBehaviour` and `ReceiverBehaviour`. Notice that neither of these classes is abstract, so they can be directly instantiated passing appropriate parameters to their constructors.

`SenderBehaviour` extends `OneShotBehaviour` and allows to send a message.

`ReceiverBehaviour` extends `Behaviour` and allows to receive a message which can be matched against a pattern; the behaviour blocks itself (without stopping all other agent activities) if no suitable messages are present in the queue. This class has been enhanced with timeout support to provide behaviour-specific timeouts, where a single `ReceiverBehaviour` can block waiting for a message and restart if nothing is received within a specific amount of time.

A more complete description of all these classes follows.

### 3.4.1 class Behaviour

Provides an abstract base class for agent behaviours, allowing behaviour scheduling independently of its actual concrete class. Moreover, it sets the basis for behaviour scheduling as it allows for state transitions (i.e. blocking and restarting a behaviour object).

The `block()` method allows to block a behaviour object until some event happens (typically, until a message arrives). This method leaves unaffected the other behaviours of an agent, thereby allowing finer grained control on agent multitasking. This method puts the behaviour in a queue of blocked behaviours and takes effect as soon as `action()` returns. All blocked behaviours are rescheduled as soon as a new message arrives. Moreover, a behaviour

object can block itself for a limited amount of time passing a timeout value to `block()` method, expressed in milliseconds. In future releases of JADE, more wake up events will be probably considered. A behaviour can be explicitly restarted by calling its `restart()` method.

Summarizing, a blocked behaviour can resume execution when one of the following three conditions occurs:

1. An ACL message is received by the agent this behaviour belongs to.
2. A timeout associated with this behaviour by a previous `block()` call expires.
3. The `restart()` method is explicitly called on this behaviour.

#### 3.4.2 class SimpleBehaviour

This abstract class models atomic behaviours that cannot be interrupted. Its `reset()` method does nothing by default, but can be overridden by user defined subclasses.

#### 3.4.3 class OneShotBehaviour

This class models atomic behaviours that must be executed only once. So, its `done()` method always returns `true`.

#### 3.4.4 class CyclicBehaviour

This class models atomic behaviours that must be executed forever. So its `done()` method always returns `false`.

#### 3.4.5 class ComplexBehaviour

This abstract class allows agent programmers to compose agent behaviours in structured trees and provides natural computation units for complex behaviours, inserting suitable breakpoints automatically. This class provides a structure for behaviour composition but lacks scheduling policies for children. Therefore is a better programming practice not to extend this class but its subclasses, that is `SequentialBehaviour` and `NonDeterministicBehaviour`. Direct `ComplexBehaviour` extension is needed only when creating new policies for children (e.g. a `PriorityBasedComplexBehaviour` should extend `ComplexBehaviour` directly).

`ComplexBehaviour` lets the programmer handle its children through `addSubBehaviour()` and `removeSubBehaviour()` methods, and provides two placeholders methods, named `preAction()` and `postAction()`. These methods can be overridden by user defined subclasses when some actions are to be executed before and after running children behaviours. An useful coding idiom is adding a `reset()` call in `postAction()` to make a given `ComplexBehaviour` restart whenever terminates, thereby turning it into a cyclic composite behaviour.

#### 3.4.6 class SequentialBehaviour

This class is a `ComplexBehaviour` that executes its sub-behaviours sequentially and terminates when all sub-behaviours are done. Use this class when a complex task can be expressed as a sequence of atomic steps (e.g. do some computation, then receive a message, then do some other computation). Use a `reset()` call within `postAction()` if endless repetition of the sequence is needed.

### 3.4.7 class NonDeterministicBehaviour

This class is a `ComplexBehaviour` that executes its sub-behaviours non deterministically and terminates when a particular condition on its sub-behaviours is met. Static Factory Methods are provided to create a `NonDeterministicBehaviour` that ends when all its sub-behaviours are done, when any one among its sub-behaviour terminates or when a user defined number  $N$  of its sub-behaviours have finished. Use this class when a complex task can be expressed as a collection of parallel alternative operations, with some kind of termination condition on the spawned subtasks. Again, use a `reset()` call within `postAction()` to obtain a continuous repetition of the task.

### 3.4.8 class SenderBehaviour

Encapsulates an atomic unit which realises the “send” action. It extends `OneShotBehaviour` class and so it is executed only once. An object with this class must be given the ACL message to send (and an optional `AgentGroup`) at construction time.

### 3.4.9 class ReceiverBehaviour

Encapsulates an atomic operation which realises the “receive” action. Its action terminates when a message is received. If the message queue is empty or there is no message matching the `MessageTemplate` parameter, `action()` method calls `block()` and returns. The received message is copied into a user specified `ACLMessage`, passed in the constructor. Two more constructors take a timeout value as argument, expressed in milliseconds; a `ReceiverBehaviour` created using one of these two constructors will terminate after the timeout has expired, whether a suitable message has been received or not. An `Handle` object is used to access the received ACL message; when trying to retrieve the message suitable exceptions can be thrown if no message is available or the timeout expired without any useful reception.

### 3.4.10 Examples

In order to explain further the previous concepts, an example is reported in the following. It illustrates the implementation of two agents that, respectively, receive and send messages. The behaviour of the `AgentSender` extend the `SimpleBehaviour` class so it simply sends some messages to the receiver and than kills itself. The `Agent Receiver` has instead a behaviour that extends `CyclicBehaviour` class and shows different kinds to receive messages.

**File AgentSender.java**

```
package examples.receivers;

import java.io.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class AgentSender extends Agent {
```

```

protected void setup() {
    addBehaviour(new SimpleBehaviour(this) {
        private boolean finished = false;
        public void action() {
            try{
                System.out.println("\nEnter responder agent name: ");
                BufferedReader buff = new BufferedReader(new
                    InputStreamReader(System.in));
                String responder = buff.readLine();
                ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                msg.addReceiver(new AID(responder));
                msg.setContent("FirstInform");
                send(msg);
                System.out.println("\nFirst INFORM sent");
                doWait(5000);
                msg.setLanguage("PlainText");
                msg.setContent("SecondInform");
                send(msg);
                System.out.println("\nSecond INFORM sent");
                doWait(5000);
                // same that second
                msg.setContent("\nThirdInform");
                send(msg);
                System.out.println("\nThird INFORM sent");
                doWait(1000);
                msg.setOntology("ReceiveTest");
                msg.setContent("FourthInform");
                send(msg);
                System.out.println("\nFourth INFORM sent");
                finished = true;
                myAgent.doDelete();
            }catch (IOException ioe){
                ioe.printStackTrace();
            }
        }
        public boolean done(){
            return finished;
        }
    });
}

```

**File AgentReceiver.java**

```
package examples.receivers;
```

```

import java.io.*;
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

public class AgentReceiver extends Agent {
    class my3StepBehaviour extends SimpleBehaviour {
        final int FIRST = 1;
        final int SECOND = 2;
        final int THIRD = 3;
        private int state = FIRST;
        private boolean finished = false;
        public my3StepBehaviour(Agent a) {
            super(a);
        }
        public void action() {
            switch (state){
                case FIRST: {if (op1())
                            state = SECOND;
                            else
                                state= FIRST;
                            break;}
                case SECOND:{op2(); state = THIRD; break;}
                case THIRD:{op3(); state = FIRST; finished = true; break;}
            }
        }

        public boolean done() {
            return finished;
        }

        private boolean op1(){
            System.out.println( "\nAgent "+getLocalName()+" in state 1.1 is
waiting for a message");
            MessageTemplate m1 =
                MessageTemplate.MatchPerformative(ACLMessage.INFORM);
            MessageTemplate m2 =
                MessageTemplate.MatchLanguage("PlainText");
            MessageTemplate m3 =
                MessageTemplate.MatchOntology("ReceiveTest");
            MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
            MessageTemplate notm3 = MessageTemplate.not(m3);

```

```

        MessageTemplate mlandm2_and_notm3 =
            MessageTemplate.and(mlandm2, notm3);

        //The agent waits for a specific message. If it doesn't arrive
        // the behaviour is suspended until a new message arrives.
        ACLMessage msg = receive(mlandm2_and_notm3);

        if (msg!= null){
            System.out.println("\nAgent "+ getLocalName() +
                " received the following message in state 1.1: " +
                msg.toString());
            return true;
        }
        else {
            System.out.println("\nNo message received in state 1.1");
            block();
            return false;
        }
    }

    private void op2(){
        System.out.println("\nAgent "+ getLocalName() + " in state 1.2
is waiting for a message");
        //Using a blocking receive causes the block
        // of all the behaviours
        ACLMessage msg = blockingReceive(5000);
        if(msg != null)
            System.out.println("\nAgent      "+      getLocalName()      +
                " received the following message in state 1.2: "
                +msg.toString());
        else
            System.out.println("\nNo message received in state 1.2");
    }

    private void op3() {
        System.out.println("\nAgent: "+getLocalName()+
            " in state 1.3 is waiting for a message");
        MessageTemplate m1 =
            MessageTemplate.MatchPerformative(ACLMessage.INFORM);
        MessageTemplate m2 = MessageTemplate.MatchLanguage("PlainText");
        MessageTemplate m3 =
            MessageTemplate.MatchOntology("ReceiveTest");
        MessageTemplate mlandm2 = MessageTemplate.and(m1,m2);
    }

```



```

        MessageTemplate mlandm2_and_m3 =
            MessageTemplate.and(mlandm2, m3);
        //blockingReceive and template
        ACLMessage msg = blockingReceive(mlandm2_and_m3);
        if (msg!= null)
            System.out.println("\nAgent      "+      getLocalName()      +
                " received the following message in state 1.3: "
                + msg.toString());
        else
            System.out.println("\nNo message received in state 1.3");
    }
} // End of my3StepBehaviour class

protected void setup() {
    my3StepBehaviour mybehaviour = new my3StepBehaviour(this);
    addBehaviour(mybehaviour);
}
}

```

### 3.5 Interaction Protocols

FIPA specifies a set of standard interaction protocols, that can be used as standard templates to build agent conversations. For every conversation among agents, JADE distinguishes the *Initiator* role (the agent starting the conversation) and the *Responder* role (the agent engaging in a conversation after being contacted by some other agent). JADE provides ready made behaviour classes for both roles in conversations following most FIPA interaction protocols. These classes can be found in `jade.proto` package, as described in this section.

All Initiator behaviours terminate and are removed from the queue of the agent tasks, as soon as they reach any final state of the interaction protocol. In order to allow the re-use of the Java objects representing these behaviours without having to recreate new objects, all initiators include a number of `reset` methods with the appropriate arguments. Furthermore, all Initiator behaviours, but `FipaRequestInitiatorBehaviour`, are 1:N, i.e. can handle several responders.

All Responder behaviours, instead, are cyclic and they are rescheduled as soon as they reach any final state of the interaction protocol. Notice that this feature allows the programmer to limit the maximum number of responder behaviours that the agent should execute in parallel. For instance, the following code ensures that a maximum of two contract-net tasks will be executed simultaneously.

```

Protected void setup() {
    addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
    addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
}

```

A complete reference for these classes, as for the classes supporting other interaction protocols, can be found in JADE HTML documentation and class reference.

The interface of these behaviours are quite homogeneous, however it is here anticipated that in the next version of JADE we intend to improve this interface.

### 3.5.1 FIPA-Request

This interaction protocol allows the Initiator to request the Responder to: execute an action and inform the initiator of the result of that action, including a possible failure. The interaction protocol is deeply described in the FIPA specifications while the following figure is just a simplification for the programmer.

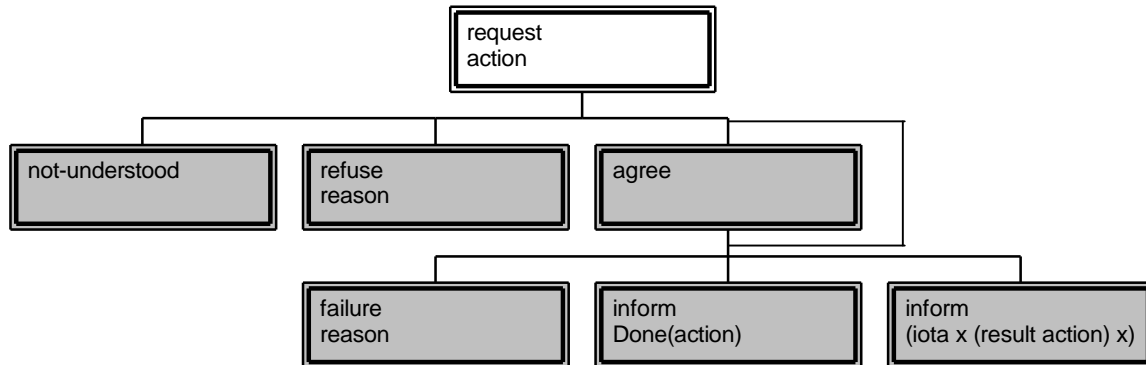


Figure 5 - FIPA-Request Interaction Protocol

#### 3.5.1.1 FipaRequestInitiatorBehaviour

Three arguments must be passed to the constructor of the `FipaRequestInitiatorBehaviour` object:

- a reference to the agent object that is needed to send and receive messages;
- the ACL message representing the request message to be sent
- and, optionally, a message template to match replies against.

The behaviour takes care of setting the message performative to "request", the protocol field to "fipa-request", and creating two appropriate values for the conversation-id and reply-with fields of the ACLMessage.

The message template argument is used to match incoming replies; replies will have to match the passed template **and** have "fipa-request" as protocol field **and** match the used "conversation-id" and ":reply-with" fields.

When a reply message is received from the responder agent, an handler method is called for the specific message kind, and the complete reply message is passed to it. These five handle methods are protected and abstract, so application programmers must override them in application specific subclasses.

The behaviour takes also care of handling timeouts in waiting for the answer. The timeout is got from the reply-by field of the passed request ACLMessage; if it was not set, then an infinite timeout is used. If the timeout expires without having received any answer, the method `handleTimeout` is called in order to give the opportunity to the programmer to decide what to do (e.g. finishing the behaviour).

#### 3.5.1.2 FipaRequestResponderBehaviour

This class implements the responder of a fipa-request interaction protocol.

Every time a message is received, whose performative is set to REQUEST and whose protocol field is set to fipa-request, this class adds a new task to the queue of the agent tasks. In practice, it does the following:

- extract the name of the action from the content of the message by calling the method `getActionName()`
- lookup for a registered Factory able to create a new `ActionHandler` to execute this action
- create the `ActionHandler` and add it to the queue of agent behaviours.

The class includes two inner classes that represent the Factory and the `ActionHandler`.

The Factory is just an interface with a single method `create(ACLMessage request)` that creates a new `ActionHandler`.

The `ActionHandler` extends the `Behaviour` class and any user-defined action handler must implement the appropriate action method for that behaviour.

Notice that in general the `FipaRequestResponderBehaviour` can be used without a need to extend the class. However, care must be taken to the default implementation of the `getActionName()` that works only for content languages and ontologies that have been registered with the agent.

### 3.5.2 FIPA-Query

This interaction protocol allows the Initiator to query the Responder (both query-if and query-ref of FIPA) and get the result of the query, including a possible failure. The interaction protocol is deeply described in the FIPA specifications while the following figure is just a simplification for the programmer.

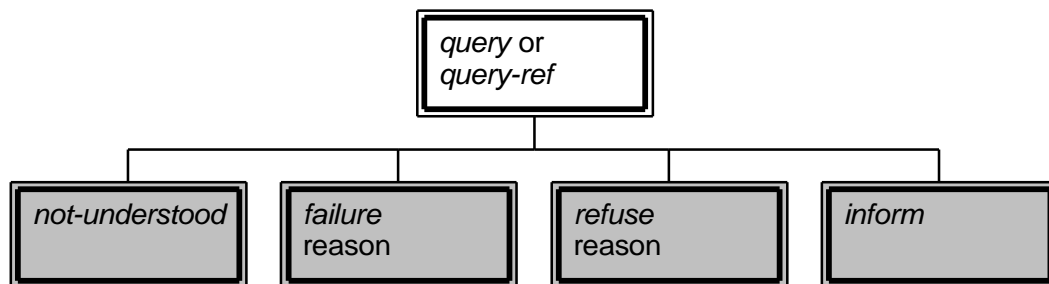


Figure 6 - FIPA-Query Interaction Protocol.

#### 3.5.2.1 FipaQueryInitiatorBehaviour

The constructor of this behaviour takes 3 parameters

```
public FipaQueryInitiatorBehaviour(Agent a, ACLMessage msg, List responders)
```

the calling agent, the query message to be sent and the group of agents to which the message should be sent. In fact, the protocol is implemented 1:N with one initiator and several responders.

In order to use correctly this behaviour, the programmer should implements a class that extends `FipaQueryInitiatorBehaviour`, create a new instance of this class and add it to the queue of the agent behaviours (via the method `addBehaviour()`). This class must implement 2 methods that are called by `FipaQueryInitiatorBehaviour`:

- `public void handleOtherMessages(ACLMessage msg)` to handle all received messages that are different from "inform" message but that still refer to this conversation;
- `public void handleInformMessages(Vector messages)` to handle the "inform" messages received in response to the query.

The behaviour takes also care of handling timeouts in waiting for the answer. The timeout is got from the `reply-by` field of the `ACLMessage` passed in the constructor; if it was not set, then an infinite timeout is used. If the timeout expires without having received any answer, the method `handleInformMessages` is executed by passing an empty vector of messages. Of course, late answers that arrive after the timeout expires are not consumed and remain in the private queue of incoming `ACLMessages`. Because this queue has a maximum size, these messages will be removed after the queue becomes full.

### 3.5.2.2 *FipaQueryResponderBehaviour*

The `FipaQueryResponderBehaviour` implements the responder role in `fipa-query` interaction protocol. The behaviour is cyclic. Its usage is the following: a class must be instantiated that extends `FipaQueryResponderBehaviour`. This new class must implement the method `handleQueryMessage()`. The instantiated class must then be added to the `Agent` object by using the method `addBehaviour()`.

The abstract method `handleQueryMessage` must be implemented by all sub-classes. The method is called whenever a new `query-if` or `query-ref` message arrives. See also the javadoc documentation.

### 3.5.3 FIPA-Contract-Net

This interaction protocol allows the Initiator to send a Call for Proposal to a set of responders, evaluate their proposals and then accept the preferred one (or even reject all of them). The interaction protocol is deeply described in the FIPA specifications while the following figure is just a simplification for the programmer.

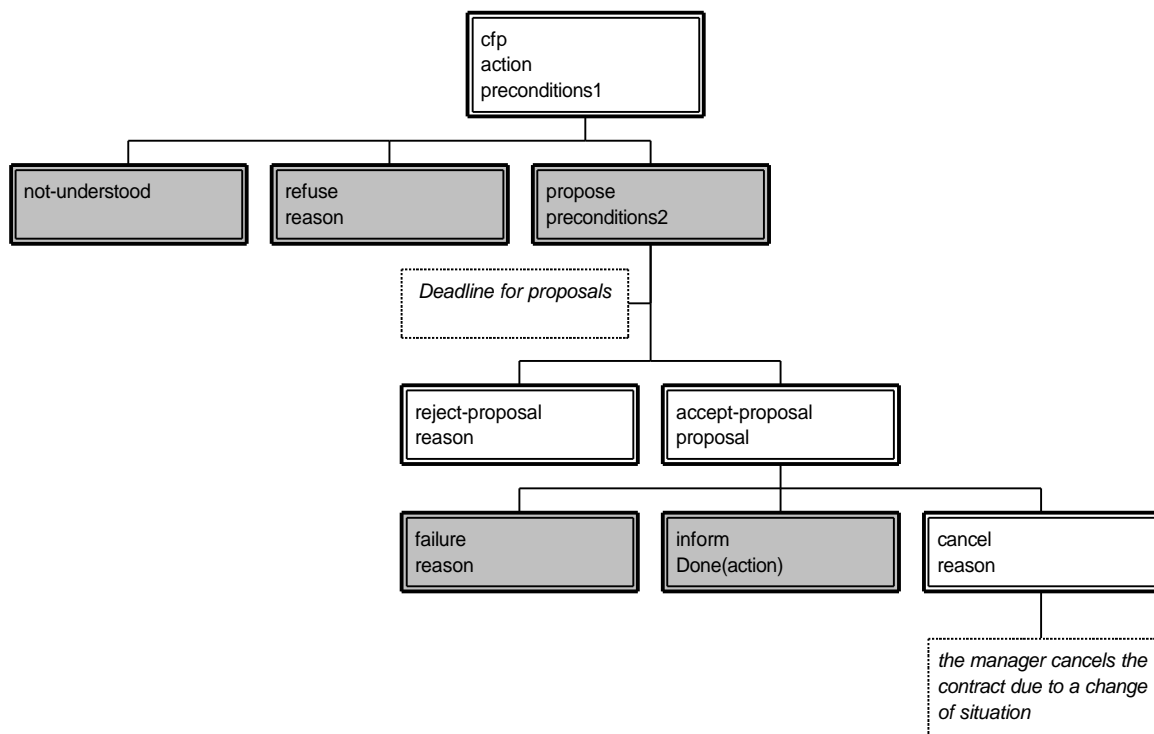


Figure 7 - FIPA-Contract-Net Interaction Protocol

### 3.5.3.1 FipaContractNetInitiatorBehaviour

This abstract behaviour implements the fipa-contract-net interaction protocol from the point of view of the agent initiating the protocol, that is the agent that sends the cfp (call for proposal) message.

The constructor of this behaviour takes 3 parameters

```
public FipaContractNetInitiatorBehaviour(Agent a, ACLMessage msg, List responders)
```

the calling agent, the CFP message to be sent and the group of agents to which the message should be sent. In fact, the protocol is implemented 1:N with one initiator and several responders.

The programmer should implement the two methods `handleProposeMessages` and `handleFinalMessages` to handle the two states of the protocol from the point of view of the initiator.

Under some circumstances, for instance when using the SL-0 content language, the content of the CFP message needs to be adapted to each receiver. For this reason, the method `createcfpcontent` is called before sending each message. The default implementation returns exactly the same content independently of the receiver; the programmer might also wish to override this default implementation.

The behaviour takes also care of handling timeouts in waiting for the answers. The timeout is got from the `reply-by` field of the `ACLMessage` passed in the constructor; if it was not set, then an infinite timeout is used. If the timeout expires without having received any answer, the method `handleXXXMessages` is executed by passing an empty vector of messages. Of course, late answers that arrive after the timeout expires are not consumed and remain in the private

queue of incoming ACLmessages. Because this queue has a maximum size, these messages will be removed after the queue becomes full.

#### 3.5.4 FipaContractNetResponderBehaviour

This abstract behaviour class implements the fipa-contract-net interaction protocol from the point of view of a responder to a call for proposal (cfp) message.

The programmer should extend this class by implementing the `handleXXX` methods that are called to handle the types of messages that can be received in this protocol.

### 3.6 Application-defined content languages and ontologies

#### 3.6.1 Rationale

When an agent A communicates with another agent B, a certain amount of information I is transferred from A to B by means of an ACL message.

Inside the ACL message I is represented as a content expression consistent with a proper content language (e.g. SL) and encoded in a proper format (e.g. string).

Both A and B have their own (possibly different) way of internally representing I.

Taking into account that the way an agent internally represent a piece information must allow an easy handling of that piece of information, it is quite clear that the representation used in an ACL content expression is not suitable for the inside of an agent.

For example the information that *the person Giovanni is 33 years old* in an ACL content expression could be represented as the string

```
(person (name Giovanni) (age 33) )
```

Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Giovanni would require each time to parse the string.

Considering software agents written in Java (as JADE's agents are), information can conveniently be represented inside an agent as Java objects.

For example representing the above information about Giovanni as an instance (an object) of an application-specific class

```
class Person{
    String name;
    int age;

    public String getName() {return name; }
    public void setName(String n) {name = n; }
    public int getAge() {return age; }
    public void setAge(int a) {age = a; }
    ...
}
```

initialized with

```
name = "Giovanni";
```

```
age = 33;
```

would allow to handle it very easily.

It is clear however that if on one hand information handling inside an agent is facilitated, on the other hand each time agent A sends a piece of information I to agent B,

- 1) A needs to convert his internal representation of I into the corresponding ACL content expression representation and B needs to perform the opposite conversion.
- 2) Moreover B should also check that I complies with the rules (i.e. for instance that the age of Giovanni is actually an integer value) of the ontology by means of which both A and B ascribe a proper meaning to I.

The support for application-defined ontology and content languages provided by JADE is designed to support agent internal representation of information as Java objects, as described above, by minimizing the developer effort in performing the above conversion and check operations.

### 3.6.2 The conversion pipeline

Each time an information has to be inserted into or extracted from an ACL content expression the JADE framework automatically performs the pipeline depicted in figure.

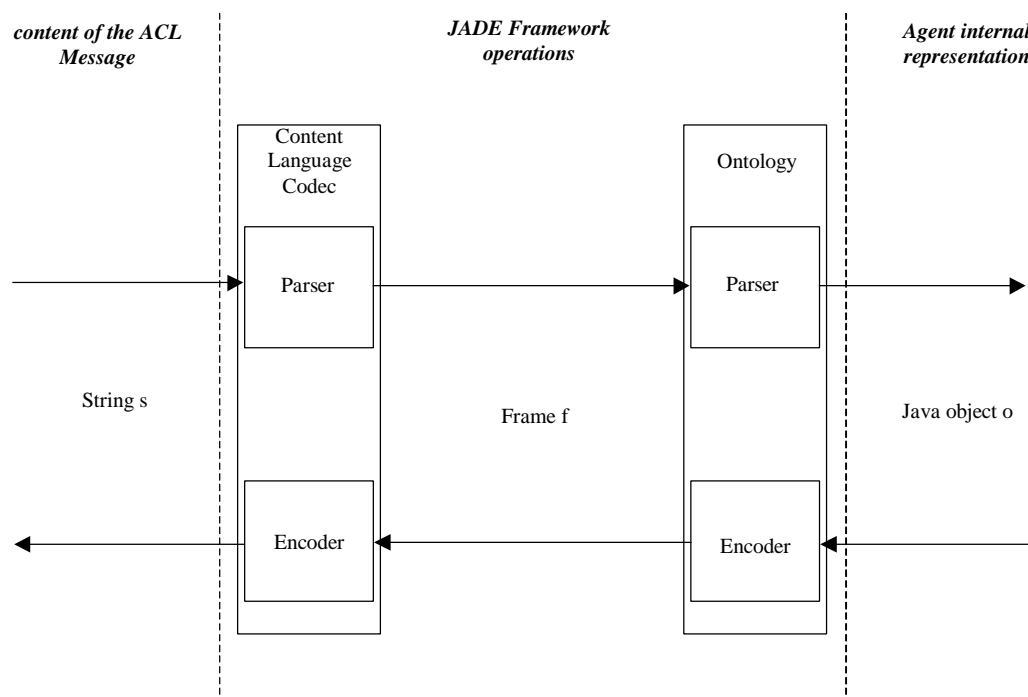


Figure 8 - Pipeline of the message content encoding/decoding

First an appropriate content language **codec** object is able to parse a content expression and to convert it into a t-uple<sup>4</sup> of Frame<sup>5</sup> objects.

<sup>4</sup> A content expression can include more than one entity in the domain of discourse. E.g. the content of a REFUSE ACL message is a t-uple with 2 elements: an action expression and the reason why the agent sending the message refuses to accomplish that action.

An appropriate **ontology** object is then able to check whether a `Frame` object is consistent with one of the schemas defining the **roles**<sup>6</sup> included in the ontology and, in this case, to convert the `Frame` object into a properly initialized instance of the application-specific class (e.g. the `Person` class mentioned above) representing the matched role.

The opposite pipeline allows to convert a sentence belonging to the domain of discourse, and represented as a Java object, into the appropriate content language and encoding.

The JADE framework hides the stages of this pipeline to the programmer who just needs to call the following methods of the `Agent` class.

```
List extractContent(ACLMessage msg);
void fillContent(ACLMessage msg, List content);
```

As already mentioned the content of an ACL message is in general a t-uple of entity inn the domain of discourse. In Java this is represented as a `List`.

The programmer however has to create and add to the resources of the agent the codec and ontology objects mentioned above as described in the followings.

### 3.6.3 Codec of a Content Language

Each content language codec in JADE must implement the interface `jade.lang.Codec` and, in particular, the two methods `decode()` and `encode()` to respectively

- parse the content in an ACL message and convert it into a `List` of `Frame` objects
- encode the content from a `List` of `Frame` objects into the content language syntax and encoding.

The `Frame` class is a neutral type (i.e. it does not distinguish between concepts, actions and predicates), that has been designed in order to allow accessing its slots both by name (e.g. (*divide :dividend 10 :divisor 2*)) and by position (e.g. (*divide 10 2*)).

This `Codec` object must then be added to the resources of each agent, which wishes to use that language, by using the method `registerLanguage()` available in the `Agent` class.

---

<sup>5</sup> The JADE Framework uses an internal representation of information based on the class `Frame`. A `Frame` object has a name and a set of slots each one being characterized by a name, a position (within the frame) and an untyped value.

Whatever entity in the domain of discourse (i.e. whatever information) can be represented as a `Frame` object.

<sup>6</sup> An ontology basically includes all the concepts, predicates and actions, collectively called **roles**, that are meaningful for the agents sharing this ontology. For instance the concepts *Company* and *Person*, the predicate *WorksFor* and the action *Engage* can be **roles** in an ontology dealing with employees. All elements in the domain of discourse (e.g. the person Giovanni) are instances of one of the **roles** composing the ontology.



By means of this operation a Codec object is associated to a content language name. When the `fillContent()` and `extractContent()` methods are called the Codec object associated to the content language indicated in the `:language` slot of the ACL message will be used to perform the conversion pipeline described in previous chapter.

Notice that JADE already includes the Codec for SL-0 (one of the standard content languages defined by FIPA) that is the class `jade.lang.sl.SL0Codec`. For an agent using SL0 it will be therefore sufficient to insert the instruction

```
registerLanguage("SL0", new SL0Codec());
```

### 3.6.4 Creating an Ontology

Each ontology in JADE must implement the `jade.onto.Ontology` interface.

It is important to note however that in the adopted approach an ontology is represented by an instance of a class implementing the `jade.onto.Ontology` interface and not just by that class.

More in details a class implementing the `jade.onto.Ontology` interface only embeds the definition of the semantic checks that will be performed when some information is received. For example an implementation can check that the age of a person is an integer value, while another implementation can also check that that integer value is  $> 0$ . All the ontological roles included in the ontology (such as the concept of person) must on the other hand be added at run-time to an instance of the above class.

Two instances `o1` and `o2` of the same class `O` implementing the `jade.onto.Ontology` interface can represent two different ontologies provided that at run-time different ontological roles are added to `o1` and `o2`.

A class, `jade.onto.DefaultOntology`, providing a default implementation of the `Ontology` interface is already provided by JADE. This is simple but still expected to be useful in most practical applications.

At the end of the day creating an ontology requires the following steps:

- Defining an application-specific class for each role in the ontology
- Creating an object of class `DefaultOntology`
- Adding to that object all the ontological roles as described below.

Each ontological role is described by a name and a number of **slots**. The `SlotDescriptor` class is provided to describe the characteristics of a slot of an ontological role.

The method `addRole()` by means of which a role is added to an ontology object takes therefore the following parameters:

- A `String` indicating the name of the added role. This parameter is missing for an unnamed slot.
- An array of `SlotDescriptor` each one describing a slot of the added role.

- A factory class of objects that are instances of the application specific class representing the added role. The JADE framework uses this factory to create proper Java objects in the conversion pipeline described above. In order for this to be carried out properly, this factory class must implement the `jade.onto.RoleEntityFactory` interface.

For example, adding the *person* role described by the `Person` class mentioned above, to a previously created ontology object `myOnto` will look like

```
Ontology myOnto = new DefaultOntology();
.....
myOnto.addRole(
    "Person",
    new SlotDescriptor[]{
        new SlotDescriptor("name", Ontology.PRIMITIVE_SLOT,
            Ontology.STRING_TYPE, Ontology.M),
        new SlotDescriptor("age", Ontology.PRIMITIVE_SLOT,
            Ontology.INTEGER_TYPE, Ontology.O)
    },
    new RoleEntityFactory() {
        public Object create (Frame f) { return new Person(); }
    }
    public Class getClassForRole() {return Person.class; }
);
```

Each slot has

- A **name** and/or a **position** (implicitly defined by the position in the array of `SlotDescriptors`) identifying the slot.
- A **category** stating that the value of the slot can be a primitive entity such as a string or an integer (`Ontology.PRIMITIVE_SLOT`), an instance of another ontological role (`Ontology.FRAME_SLOT`) or a set (`Ontology.SET_SLOT`) or sequence (`Ontology.SEQUENCE_SLOT`) of entities.
- A **type** defining the primitive type (for primitive slots) or role (for frame slots) of the value of the slot or of the elements in the set/sequence in case of set slots or sequence slots.
- A **presence** flag defining whether the slot is mandatory (`Ontology.M`) or optional (`Ontology.O`).

In the above case the *person* role has two named slots called *name* and *age*. The first is mandatory (an exception will be thrown if this slot has a null value) and permitted values are of type `String`. The second is optional and permitted values are of type `Integer`.

As a further example three other roles are added to the ontology represented by the `myOnto` object.

- *Address*, with three named slots, *street*, *number* and *city*, of type `String`, `Integer` and `String` respectively and all mandatory.

- *Company* with two named slots, *name* and *address*, of type *String* and *Address* (i.e. the values of this slot are instances of the *Address* role) respectively, one mandatory and the other optional.
- *Engage* (the action of engaging a person in a company) with two unnamed slots of type *Person* and *Company* respectively and both mandatory.

Application specific class representing the *Address* role

```
class Address {
    private String street;
    private Integer number;
    private String city;

    public String getStreet() { return street; }
    public void setStreet(String s) { street = s; }
    public Integer getNumber() { return number; }
    public void setNumber(Integer n) { number = n; }
    public String getCity() { return city; }
    public void setCity(String c) { city = c; }
}
```

Application specific class representing the *Company* role

```
class Company {
    private String name;
    private Address address;

    public void setName(String n) { name = n; }
    public String getName() { return name; }
    public void setAddress(Address a) { address = a; }
    public Address getAddress() { return address; }
}
```

Application specific class representing the *Engage* role

```
class Engage {
    private Person personToEngage;
    private Company engager;

    public void set_0(Person p) { personToEngage = p; }
    public Person get_0() { return personToEngage; }
    public void set_1(Company c) { engager = c; }
    public Company get_1() { return engager; }
}
```

Code for adding the *Address*, *Company* and *Engage* roles to the ontology.

```
myOnto.addRole(
    "Address",
    new SlotDescriptor[]{
        new SlotDescriptor("street", Ontology.PRIMITIVE_SLOT,
```

```

        Ontology.STRING_TYPE, Ontology.M),
        new SlotDescriptor("number", Ontology.PRIMITIVE_SLOT,
        Ontology.INTEGER_TYPE, Ontology.M)
        new SlotDescriptor("city", Ontology.PRIMITIVE_SLOT,
        Ontology.STRING_TYPE, Ontology.M)
    },
    new RoleEntityFactory() {
        public Object create (Frame f) { return new Address();
    }
        public Class getClassForRole() {return Address.class;
    }
    }
);

myOnto.addRole(
    "Company",
    new SlotDescriptor[]{
        new SlotDescriptor("name", Ontology.PRIMITIVE_SLOT,
        Ontology.STRING_TYPE, Ontology.M),
        new SlotDescriptor("address", Ontology.FRAME_SLOT,
        "Address" ,Ontology.O)
    },
    new RoleEntityFactory() {
        public Object create (Frame f) { return new Company();
    }
        public Class getClassForRole() {return Company.class;
    }
    }
);

myOnto.addRole(
    "engage",
    new SlotDescriptor[]{
        new SlotDescriptor(Ontology.FRAME_SLOT, "Person"
        Ontology.M),
        new SlotDescriptor(Ontology.FRAME_SLOT, "Company",
        Ontology.M)
    },
    new RoleEntityFactory() {
        public Object create (Frame f) { return new Engage();
    }
        public Class getClassForRole() {return Engage.class; }
    }
);

```

```
);
```

The ontology object must finally be added to the resources of each agent, which wishes to use that ontology, by using the method `registerOntology()` available in the `Agent` class.

By means of this operation an `Ontology` object is associated to a name. When the `fillContent()` and `extractContent()` methods are called the `Ontology` object associated to the content language indicated in the `:ontology` slot of the ACL message will be used to perform the conversion pipeline initially described.

### 3.6.5 Application specific classes representing ontological roles

In order to represent an ontological role (i.e. in order to be accepted by the `Ontology` object), a Java class must obey to some rules:

1) For each slot in the represented role named `XXX`, of category `Ontology.PRIMITIVE_SLOT` or `Ontology.FRAME_SLOT` and of type `T` the class must have two accessible methods with the following signature:

```
public T getXXX();
public void setXXX(T t);
```

2) For each slot in the represented role named `XXX`, of category `Ontology.SET_SLOT` or `Ontology.SEQUENCE_SLOT` and with elements of type `T`, the class must have two accessible methods with the following signature:

```
public Iterator getAllXXX();
public void addXXX(T t);
```

3) For each unnamed slot use “\_p” (being `p` the position of the slot) instead of the slot name for the get and set methods (see the `Engage` class mentioned above for an example).

4) In all previous cases the type `T` cannot be a primitive type such as `int`, `float` or `boolean`. Use `Integer`, `Float`, `Boolean` .... instead.

### 3.6.6 Discovering the ontological role of a Java object representing an entity in the domain of discourse

As already mentioned, when an ACL message is received, provided that the proper ontology and content language codec objects has been previously registered, the content of the ACL message can be easily converted into a list of proper Java objects by means of the `extractContent()` method.

```
List l = extractContent( msg );
```

In general however the receiving agent does not know a-priori the role of each Java object in the list. In order to discover it the ontology object must be used as described in the example below referring to the first object in the list.

```
Object obj = l.get(0);
Ontology onto = lookupOntology(msg.getOntology());
String roleName = onto.getRoleName(obj.getClass());
```

The `lookupOntology()` is a method of the `Agent` class that returns the ontology object previously associated to a given name by calling the `registerOntology()` method.

Once discovered the role of the entity represented by an object it will be possible to cast it to the application specific class representing that role.

### 3.6.7 Setting and getting the content of an ACL message.

Having registered a content language codec and an ontology with the agent, it is possible to exploit the automatic support of the JADE framework to set and get the content of an ACL message. The `Agent` class provides two methods for this purpose: *extractContent* and *fillContent* to implement the parsing, respectively, encoding operations shown in the pipeline figure.

The first method extracts the content from an ACL message and returns a List of Java objects (one object for each element of the t-uple in the content) by calling the appropriate content language Codec (according to the value of the *:language* parameter of the ACL message) and the appropriate Ontology (according to the value of the *:ontology* parameter of the ACL message).

The second method, instead, makes the opposite operation, that is it fills in the content of an ACL message by interpreting a List of Java objects with the appropriate Ontology and content language Codec, as specified by the values of the *:ontology* and the *:language* parameter of the ACL message.

Refer to the javadoc documentation for a detailed description of the usage of these two methods.

## 3.7 Support for Agent Mobility

Using JADE, application developers can build mobile agents, which are able to migrate or copy themselves across multiple network hosts. In this version of JADE, only *intra-platform* mobility is supported, that is a JADE mobile agent can navigate across different agent containers but it is confined to a single JADE platform.

Moving or cloning is considered a state transition in the life cycle of the agent. Just like all the other life cycle operation, agent motion or cloning can be initiated either by the agent itself or by the AMS. The `Agent` class provides a suitable API, whereas the AMS agent can be accessed via FIPA ACL as usual.

Mobile agents need to be *location aware* in order to decide when and where to move. Therefore, JADE provides a proprietary ontology, named *jade-mobility-ontology*, holding the necessary concepts and actions. This ontology is contained within the `jade.domain.MobilityOntology` class, and it is an example of the new application-defined ontology support.

### 3.7.1 JADE API for agent mobility.

The two public methods `doMove()` and `doClone()` of the `Agent` class allow a JADE agent to migrate elsewhere or to spawn a remote copy of itself under a different name. Method `doMove()` takes a `jade.core.Location` as its single parameter, which represents the intended destination for the migrating agent. Method `doClone()` also takes a `jade.core.Location` as parameter, but adds a `String` containing the name of the new agent that will be created as a copy of the current one.

Looking at the documentation, one finds that `jade.core.Location` is an abstract interface, so application agents are not allowed to create their own locations. Instead, they must ask the AMS for the list of the available locations and choose one. Alternatively, a JADE agent can also request the AMS to tell where (at which location) another agent lives.

Moving an agent involves sending its code and state through a network channel, so user defined mobile agents must manage the serialization and unserialization process. Some among the various resources used by the mobile agent will be moved along, while some others will be disconnected before moving and reconnected at the destination (this is the same distinction between transient and non-transient fields used in the *Java Serialization API*). JADE makes available a couple of matching methods in the `Agent` class for resource management.

For agent migration, the `beforeMove()` method is called at the starting location just before sending the agent through the network (with the scheduler of behaviours already stopped), whereas the `afterMove()` method is called at the destination location as soon as the agent has arrived and its identity is in place (but the scheduler has not restarted yet).

For agent cloning, JADE supports a corresponding method pair, the `beforeClone()` and `afterClone()` methods, called in the same fashion as the `beforeMove()` and `afterMove()` above. The four methods above are all protected methods of the `Agent` class, defined as empty placeholders. User defined mobile agents will override the four methods as needed.

### 3.7.2 JADE Mobility Ontology.

The *jade-mobility-ontology* ontology contains all the concepts and actions needed to support agent mobility. JADE provides the class `jade.domain.MobilityOntology`, working as a *Singleton* and giving access to a single, shared instance of the JADE mobility ontology through the `instance()` method.

The ontology contains ten frames (six concepts and four actions), and a suitable inner class is associated with each frame using a `RoleFactory` object (see Section **Error! Reference source not found.** for details). The following list shows all the frames and their structure.

- `Mobile-agent-description`; describes a mobile agent going somewhere. It is represented by the `MobilityOntology.MobileAgentDescription` inner class.

Slot Name	Slot Type	Mandatory/Optional
<b>name</b>	<b>AID</b>	<b>Mandatory</b>
<b>destination</b>	<b>Location</b>	<b>Mandatory</b>
agent-profile	mobile-agent-profile	Optional
agent-version	String	Optional
signature	String	Optional

- `mobile-agent-profile`; describes the computing environment needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentProfile` inner class.

Slot Name	Slot Type	Mandatory/Optional
<code>system</code>	<code>mobile-agent-system</code>	Optional
<code>language</code>	<code>mobile-agent-language</code>	Optional
<b><code>os</code></b>	<b><code>Mobile-agent-os</code></b>	<b>Mandatory</b>

- `mobile-agent-system`; describes the runtime system used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentSystem` inner class.

Slot Name	Slot Type	Mandatory/Optional
<b><code>name</code></b>	<b><code>String</code></b>	<b>Mandatory</b>
<b><code>major-version</code></b>	<b><code>Long</code></b>	<b>Mandatory</b>
<code>minor-version</code>	<code>Long</code>	Optional
<code>dependencies</code>	<code>String</code>	Optional

- `mobile-agent-language`; describes the programming language used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentLanguage` inner class.

Slot Name	Slot Type	Mandatory/Optional
<b><code>name</code></b>	<b><code>String</code></b>	<b>Mandatory</b>
<b><code>major-version</code></b>	<b><code>Long</code></b>	<b>Mandatory</b>
<code>minor-version</code>	<code>Long</code>	Optional
<code>dependencies</code>	<code>String</code>	Optional

- `mobile-agent-os`; describes the operating system needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentOS` inner class.

Slot Name	Slot Type	Mandatory/Optional
<b><code>name</code></b>	<b><code>String</code></b>	<b>Mandatory</b>
<b><code>major-version</code></b>	<b><code>Long</code></b>	<b>Mandatory</b>
<code>minor-version</code>	<code>Long</code>	Optional
<code>dependencies</code>	<code>String</code>	Optional

- `Location`; describes a location where an agent can go. It is represented by the `MobilityOntology.Location` inner class.

Slot Name	Slot Type	Mandatory/Optional
<b><code>name</code></b>	<b><code>String</code></b>	<b>Mandatory</b>
<b><code>transport-</code></b>	<b><code>String</code></b>	<b>Mandatory</b>



protocol		
transport-address	String	Mandatory

- ❑ `move-agent`; the action of moving an agent from a location to another. It is represented by the `MobilityOntology.MoveAction` inner class.

This action has a single, unnamed slot of type `mobile-agent-description`. The argument is mandatory.

- ❑ `clone-agent`; the action performing a copy of an agent, possibly running on another location. It is represented by the `MobilityOntology.CloneAction` inner class.

This action has two unnamed slots: the first one is of `mobile-agent-description` type and the second one is of `String` type. Both arguments are mandatory.

- ❑ `where-is-agent`; the action of requesting the location where a given agent is running. It is represented by the `MobilityOntology.WhereIsAgent` inner class.

This action has a single, unnamed slot of type `AID`. The argument is mandatory.

- ❑ `query-platform-locations`; the action of requesting the list of all the platform locations. It is represented by the `MobilityOntology.QueryPlatformLocations` inner class.

This action has no slots.

**Notice that this ontology has no counter-part in any FIPA specifications. It is intention of the JADE team to update the ontology as soon as a suitable FIPA specification will be available.**

### 3.7.3 Accessing the AMS for agent mobility.

The JADE AMS has some extensions that support the agent mobility, and it is capable of performing all the four actions present in the *jade-mobility-ontology*. Every mobility related action can be requested to the AMS through a *FIPA-request* protocol, with *jade-mobility-ontology* as ontology value and *FIPA-SLO* as language value.

The `move-agent` action takes a `mobile-agent-description` as its parameter. This action moves the agent identified by the name and address slots of the `mobile-agent-description` to the location present in the destination slot.

For example, if an agent wants to move the agent *Peter* to the location called *Front-End*, it must send to the AMS the following ACL request message:

```
( REQUEST
  :sender (Agent-Identifier :name da0)
  :receiver (set (Agent-Identifier :name ams))
  :content (
    ( action (Agent-Identifier :name ams) ( move-agent
      ( mobile-agent-description
        :name (Agent-Identifier :name Peter)
```

```

        :destination
        ( Location
          :name Front-End
          :transport-protocol JADE-IPMT
          :transport-address IOR:000...Front-End
        )
      )
    )
  ) )
:language FIPA-SL0
:ontology jade-mobility-ontology
:protocol fipa-request
)

```

In the above message, the actual platform IOR has been shorted; in the actual ACL, the complete IOR was present both in the address slot for the mobile-agent-description and in the transport-address slot for the location.

The above message was written by hand and sent to the AMS using JADE *DummyAgent*. For user defined agents, a better approach is to exploit the ontological classes, exploiting the techniques described in section **Error! Reference source not found.**

A generic agent can create a new `MobilityOntology.MoveAction` object, fill its argument with a suitable `MobilityOntology.MobileAgentDescription` object, filled in turn with the name and address of the agent to move and with the `MobilityOntology.Location` object for the destination. Then, a single call to the `Agent.fillContent()` method can turn the `MoveAction` Java object into a `String` and write it into the content slot of a suitable request ACL message.

The `clone-agent` action works in the same way, but has an additional `String` argument to hold the name of the new agent resulting from the cloning process.

The `where-is-agent` action has a single AID argument, holding the identifier of the agent to locate. This action has a result, namely the location for the agent, that is put into the content slot of the `inform` ACL message that successfully closes the protocol.

For example, the request message to ask for the location where the agent *Peter* resides would be:

```

( REQUEST
  :sender (Agent-Identifier :name da0)
  :receiver (set (Agent-Identifier :name ams))
  :content ( ( action (Agent-Identifier :name ams)
                ( where-is-agent (Agent-Identifier :name Peter ) )))
  :language FIPA-SL0
  :ontology jade-mobility-ontology
  :protocol fipa-request
)

```

The resulting `Location` would be contained within an `inform` message like the following:

```
( INFORM
:sender (Agent-Identifier :name ams)
:receiver (set (Agent-Identifier :name da0))
:content ( (Result
            (action (Agent-Identifier :name ams)
              ( where-is-agent (Agent-Identifier :name Peter )))
            (set
              (Location
                :name Front-End
                :transport-protocol JADE-IPMT
                :transport-address IOR:000....Front-End
              )))
          )
:language FIPA-SL0
:ontology jade-mobility-ontology
:protocol fipa-request
)
```

The `query-platform-locations` action takes no arguments, but its result is a set of all the `Location` objects available in the current JADE platform. The message for this action is very simple:

```
( REQUEST
:sender (agent-identifier :name Johnny)
:receiver (set (Agent-Identifier :name AMS))
:content (( action (agent-identifier :name AMS)
              ( query-platform-locations ) ))
:language FIPA-SL0
:ontology jade-mobility-ontology
:protocol fipa-request
)
```

If the current platform had three containers, the AMS would send back the following inform message:

```
( INFORM
:sender (Agent-Identifier :name AMS)
:receiver (set (Agent-Identifier :name Johnny))
:content (( Result ( action (agent-identifier :name AMS)
                          ( query-platform-locations ) )
                  (set (Location
                        :name Container-1
                        :transport-protocol JADE-IPMT
                        :transport-address IOR:000....Container-1 )
                      (Location
                        :name Container-2
                        :transport-protocol JADE-IPMT
                        :transport-address IOR:000....Container-2 )
                      )
                )
          )
)
```

```

        (Location
          :name Container-3
          :transport-protocol JADE-IPMT
          :transport-address IOR:000....Container-3 )
      )))
:language FIPA-SL0
:ontology jade-mobility-ontology
:protocol fipa-request
)

```

The `MobilityOntology.Location` class implements `jade.core.Location` interface, so that it can be passed to `Agent.doMove()` and `Agent.doClone()` methods. A typical behaviour pattern for a JADE mobile agent will be to ask the AMS for locations (either the complete list or through one or more `where-is-agent` actions); then the agent will be able to decide if, where and when to migrate.

---

#### 4 A SAMPLE AGENT SYSTEM

---

We are presenting an example of an agent system explaining how to use the features available in JADE framework. In particular we will show the possibility of organising the behaviour of a single agent in different sub-behaviours and how the message exchange among agents takes place.

The agent system, in the example, is made of two agents communicating through FIPA request protocol.

*This section is still to do. Please refer to JADE examples present in `src/examples` directory. Refer also to the `README` file in `src/examples` directory to get some explanations of each example program.*

---

#### 5 RUNNING THE AGENT PLATFORM

---

##### 5.1 Software requirements

The only software requirement to execute the system is the Java Run Time Environment version 1.2

Further to the Java Compiler version 1.2, to build the system, the JavaCC parser generator (version 0.8pre or version 1.1; available from <http://www.metamata.com>), and the IDL to Java translator `idltojava` (available from the Sun Developer Connection) are also needed. However, pre-built IDL stubs and Java parser classes are included with the JADE source distribution such that the Java compiler is sufficient to build the full system.

##### 5.2 Getting the software

All the software is distributed under the LGPL license limitations. It can be downloaded from the JADE web site <http://sharon.cselt.it/projects/jade>. Five compressed files are available:

1. the source code of JADE
2. the source code of the examples

3. the documentation, including the javadoc of the JADE API and this programmer's guide
4. the binary of JADE, i.e. the jar files with all the Java classes
5. a full distribution with all the previous files

### 5.3 Running JADE from the binary distribution

Having uncompressed the archive file, a directory tree is generated whose root is `jade` and with a `lib` subdirectory. This subdirectory contains some JAR files that have to be added to the `CLASSPATH` environment variable.

Having set the classpath, the following command can be used to launch the main container of the platform. The main container is composed of the DF agent, the AMS agent, an RMI registry (that is used by JADE for intra-platform communication), and the ACC.

```
java jade.Boot [options] [Agent list]
```

Additional agent containers can be then launched on the same host, or on remote hosts, that connect themselves with the main container of the Agent Platform, resulting in a distributed system that seems a single Agent Platform from the outside.

An Agent Container can be started using the command:

```
java jade.Boot -container [options] [Agent list]
```

An alternative way of launching JADE is to use the following command, that does not need to set the `CLASSPATH`:

```
java -jar lib\jade.jar [options] [Agent list]
```

#### 5.3.1 Options available from the command line

*-h* this option prints on the standard output some help on the available options

*-container* this option launches a container that try to connect with a main container

*-host <hostname>* should be used to specify the host where the main container has been launched; the default is the local host.

*-port <portnumber>* this option allows to specify the port number where the RMI registry should be created (for the main-container) / located (for the ordinary containers)

*-conf* this option displays a graphical interface that allows to load/save the JADE configuration from a file

*-conf <fileName>* launches JADE with the options specified in the `fileName`.

*-gui* this option launches a Remote Monitoring Agent, a graphical interface that allows to monitor the agent platform (see section 6).

*-version* prints on the standard output versioning information of JADE.

#### 5.3.2 Launching agents from the command line

A list of agents can be launched directly from the command line. As described above, the `[Agent list]` part of the command is a sequence of strings separated by a space.

Each string is broken in two parts separated by a colon ':' character. The substring before the colon is taken as the agent name, whereas the substring after the colon is the name of the Java class implementing the agent. This class will be dynamically loaded by the Agent Container.

For example, a string `Peter:myAgent` means "create a new agent named Peter whose implementation is an object of class `myAgent`". The name of the class must be fully qualified, (e.g. `Peter:myPackage.myAgent`) and will be searched for according to `CLASSPATH` definition.

### 5.3.3 Example

First of all set the `CLASSPATH` to include the JAR files in the `lib` subdirectory and the current directory. For instance, for Windows 9x/NT use the following command:

```
set CLASSPATH=%CLASSPATH%;.;c:\jade\lib\jade.jar;
c:\jade\lib\jadeTools.jar; c:\jade\lib\Base64.jar
```

Execute the following command to start the main-container of the platform. Let's suppose that the hostname of this machine is "kim.cselt.it"

```
prompt> java jade.Boot -name facts -gui
```

Execute the following command to start an agent container on another machine, by telling it to join the AgentPlatform, called "facts" running on the host "kim.cselt.it", and start one agent (you must download and compile the examples agents to do that):

```
prompt> java jade.Boot -name facts -host kim.cselt.it
sender1:examples.receivers.AgentSender
```

where "sender1" is the name of the agent, while `examples.receivers.AgentSender` is the code that implements the agent.

Execute the following command on a third machine to start another agent container telling it to join the Agent Platform, called "facts" running on the host "kim.cselt.it", and then start two agents.

```
prompt> java jade.Boot -name facts -host kim.cselt.it
receiver2:examples.receivers.AgentReceiver
sender2:examples.receivers.AgentSender
```

where the agent named `sender2` is implemented by the class `examples.receivers.AgentSender`, while the agent named `receiver2` is implemented by the class `examples.receivers.AgentReceiver`.

## 5.4 Building JADE from the source distribution

If you downloaded JADE in source form and want to compile it, you basically have two methods: either you use the provided makefiles (for GNU make), or you run the Win32 .BAT files that you find in the root directory of the package. Of course, using makefiles yields more flexibility because they just build what is needed; JADE makefiles have been tested under Sun Solaris 7 with JDK 1.2.0 and under Linux under JDK 1.2.2 RC4. The batch files have been tested under Windows NT 4.0 and under Windows 95, both with JDK 1.2.2.

### 5.4.1 Building the JADE framework

If you use the makefiles, just type

```
make all
```

in the root directory; if you use the batch files, type

```
makejade
```

in the root directory. Beware that the batch file will not be able to check whether IDL stubs and parser classes already exist, so either you have `idltojava` and `JavaCC` installed, or you comment out them in the batch file.

You will end up with all JADE classes in a `classes` subdirectory. You can add that directory to your `CLASSPATH` and make sure that everything is OK by running JADE, as described in the previous section.

#### 5.4.2 Building JADE libraries

With makefiles, type

```
make lib
```

With batch files, type

```
makelib
```

This will remove the content of the `classes` directory and will create some JAR files in the `lib` directory. These JAR files are just the same you get from the binary distribution. See section 5.3 for a description on how to run JADE when you have built the JAR files. Beware that, both with makefiles and batches, you must first build the classes and then the libraries, otherwise you will end up with empty JAR files.

#### 5.4.3 Building JADE HTML documentation

With makefiles, type

```
make doc
```

With batch files, type

```
makedoc
```

You will end up with Javadoc generated HTML pages, integrated within the overall documentation. Beware that the Programmer's Guide is a PDF file that cannot be generated at your site, but you must download it (it is, of course, in the JADE documentation distribution).

#### 5.4.4 Building JADE examples and demo application

If you downloaded the examples/demo archive and have unpacked it within the same source tree, you will have to set your `CLASSPATH` to contain either the `classes` directory or the JAR files in the `lib` directory, depending on your JADE distribution, and then type:

```
make examples
```

with makefiles, or

```
makeexamples
```

with batch files.

In order to compile the Jess-based example, it is necessary to have the JESS system and to set the `CLASSPATH` to include it. The example can be compiled by typing:

```
make jesseexample
```

with makefiles, or

```
makejesseexample
```

with batch files.

#### 5.4.5 Cleaning up the source tree

If you type  
`make clean`  
 with makefiles, or if you type  
`clean`

with batch files, you will remove all generated files (classes, HTML pages, JAR files, etc.) from the source tree. If you use makefiles, you will find some other make targets you can use. Feel free to try them, especially if you are modifying JADE source code, but be aware that these other make targets are for internal use only, so they have not been documented.

### 5.5 IIOP support and inter-platform messaging

JADE supports FIPA compliant IIOP communication for inter-platform agent communication. This mechanism is used both to communicate with another JADE platform and with a non-JADE platform. JADE achieves complete transparency in message passing even when multiple agent platforms are involved, so agent developers need not worry about IIOP: JADE selects local Java events, RMI or CORBA/IIOP automatically on behalf of the application.

The only issue application developers and platform administrators must be aware-of is agent naming. An agent identifier, in fact, must include a set of URL representing the addresses where it can be contacted.

Every JADE agent inherits the addresses of its platform and its AID (including the addresses) is fully generated automatically by JADE when messages are sent externally to the platform.

Because most of the CORBA ORB implementations (including the one used by JADE) do not yet allow to choose meaningful words as object keys, JADE resorts to the alternate naming scheme, adopted also by FIPA 98 version 2.0, using OMG standard stringified IOR as agent addresses. A valid agent address can be both an URL like `iiop://fipa.org:50/acc` and an IOR as `IOR:0000000000000001649444c644f4...`

The IOR-based representation and the URL-based one are exactly equivalent, the URL being far more readable for humans than the IOR. JADE generates IOR-based addresses but can also deal with URL-based ones as long as the URL contains only printable characters (i.e. has been created by an ORB allowing explicit object key assignment) that can be parsed by a FIPA-compliant parser.

When starting up, JADE platform prints its IOR both on the standard output and in a ASCII file named *JADE.IOR*, located in the current directory; the URL for the platform (containing a binary string in the file part) is also written to the file *JADE.URL* in the current directory. Every agent address automatically includes this platform IOR that should be distributed to remote platforms in order to allow remote agents to send messages to your JADE agents. The distribution mechanism is not specified by FIPA and is application dependent.

---

## 6 GRAPHICAL USER INTERFACE TO MANAGE AND MONITOR THE AP ACTIVITY

---

To support the difficult task of debugging multi-agent applications, three tools have been developed. Each tool is packaged as an agent itself, obeying the same rules, the same communication capabilities, and the same life cycle of a generic application agent.



## 6.1 Remote Monitoring Agent

The Remote Monitoring Agent (RMA) allows controlling the life cycle of the agent platform and of all the registered agents. The distributed architecture of JADE allows also remote controlling, where the GUI is used to control the execution of agents and their life cycle from a remote host.

An RMA is a Java object, instance of the class `jade.tools.rma.rma` and can be launched from the command line as an ordinary agent (i.e. with the command `java jade.Boot myConsole:jade.tools.rma.rma`), or by supplying the `-gui` option the command line parameters (i.e. with the command `java jade.Boot -gui`).

More than one RMA can be started on the same platform as long as every instance has a different local name, but only one RMA can be executed on the same agent container.

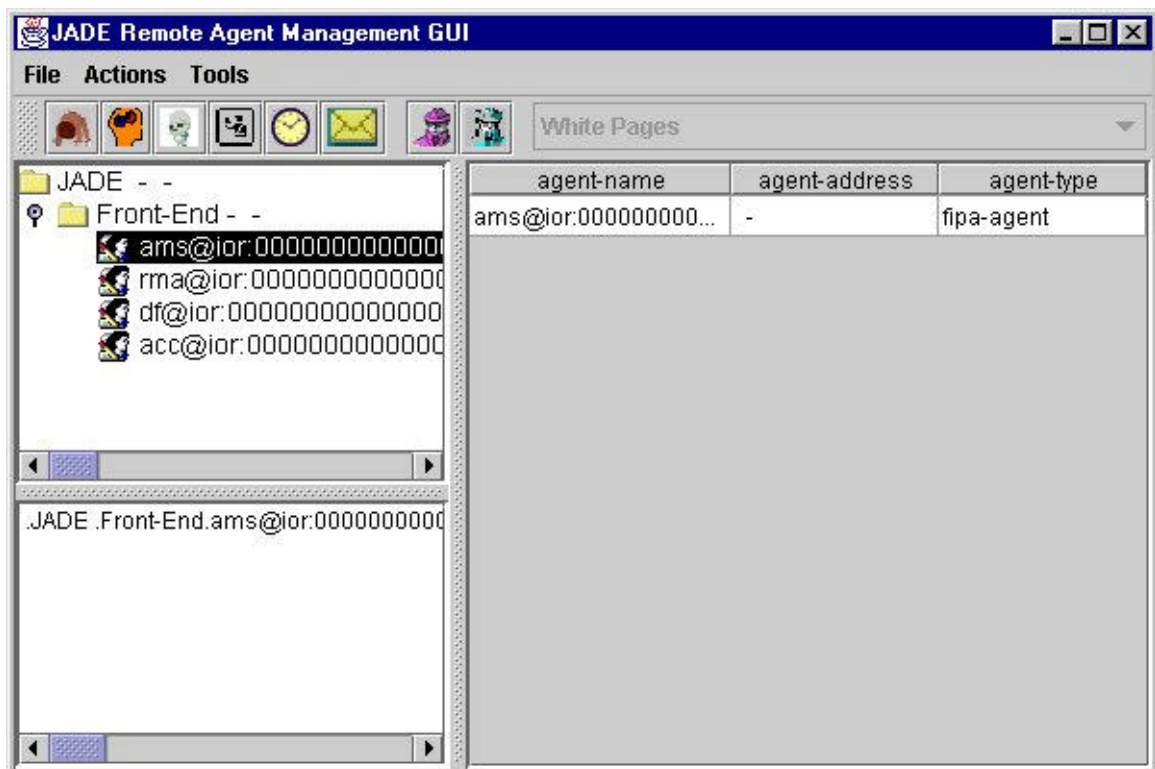


Figure 9 - Snapshot of the RMA GUI

The followings are the commands that can be executed from the menu bar (or the tool bar) of the RMA GUI.

- ◆ File menu:

This menu contains the general commands to the RMA.

- ◆ Close RMA Agent

Terminates the RMA agent by invoking its `doDelete()` method. The closure of the RMA window has the same effect as invoking this command.

- ◆ Exit this Container

Terminates the agent container where the RMA is living in, by killing the RMA and all the other agents living on that container. If the container is the Agent Platform Main-Container, then the whole platform is shut down.

- ◆ Shut down Agent Platform

Shut down the whole agent platform, terminating all connected containers and all the living agents.

- ◆ Actions menu:

This menu contains items to invoke all the various administrative actions needed on the platform as a whole or on a set of agents or agent containers. The requested action is performed by using the current selection of the agent tree as the target; most of these actions are also associated to and can be executed from toolbar buttons.

- ◆ Start New Agent

This action creates a new agent. The user is prompted for the name of the new agent and the name of the Java class the new agent is an instance of. Moreover, if an agent container is currently selected, the agent is created and started on that container; otherwise, the user can write the name of the container he wants the agent to start on. If no container is specified, the agent is launched on the Agent Platform Main-Container.

- ◆ Kill Selected Items

This action kills all the agents and agent containers currently selected. Killing an agent is equivalent to calling its `doDelete()` method, whereas killing an agent container kills all the agents living on the container and then de-registers that container from the platform. Of course, if the Agent Platform Main-Container is currently selected, then the whole platform is shut down.

- ◆ Suspend Selected Agents

This action suspends the selected agents and is equivalent to calling the `doSuspend()` method. Beware that suspending a system agent, particularly the AMS, deadlocks the entire platform.

- ◆ Resume Selected Agents

This action puts the selected agents back into the `AP_ACTIVE` state, provided they were suspended, and works just the same as calling their `doActivate()` method.

- ◆ Send Custom Message to Selected Agents

This action allows to send an ACL message to an agent. When the user selects this menu item, a special dialog is displayed in which an ACL message can be composed and sent, as shown in the figure.

- ◆ Tools menu:

This menu contains the commands to start all the tools provided by JADE to application programmers. These tools will help developing and testing JADE based agent systems.

## 6.2 DummyAgent

The DummyAgent tool allows users to interact with JADE agents in a custom way. The GUI allows to compose and send ACL messages and maintains a list of all ACL messages sent and

received. This list can be examined by the user and each message can be viewed in detail or even edited. Furthermore, the message list can be saved to disk and retrieved later. Many instances of the DummyAgent can be started as and where required.

The DummyAgent can both be launched from the Tool menu of the RMA and from the command line, as follows:

```
Java jade.Boot theDummy:jade.tools.DummyAgent.DummyAgent
```

The figure is a snapshot of the DummyAgent GUI.

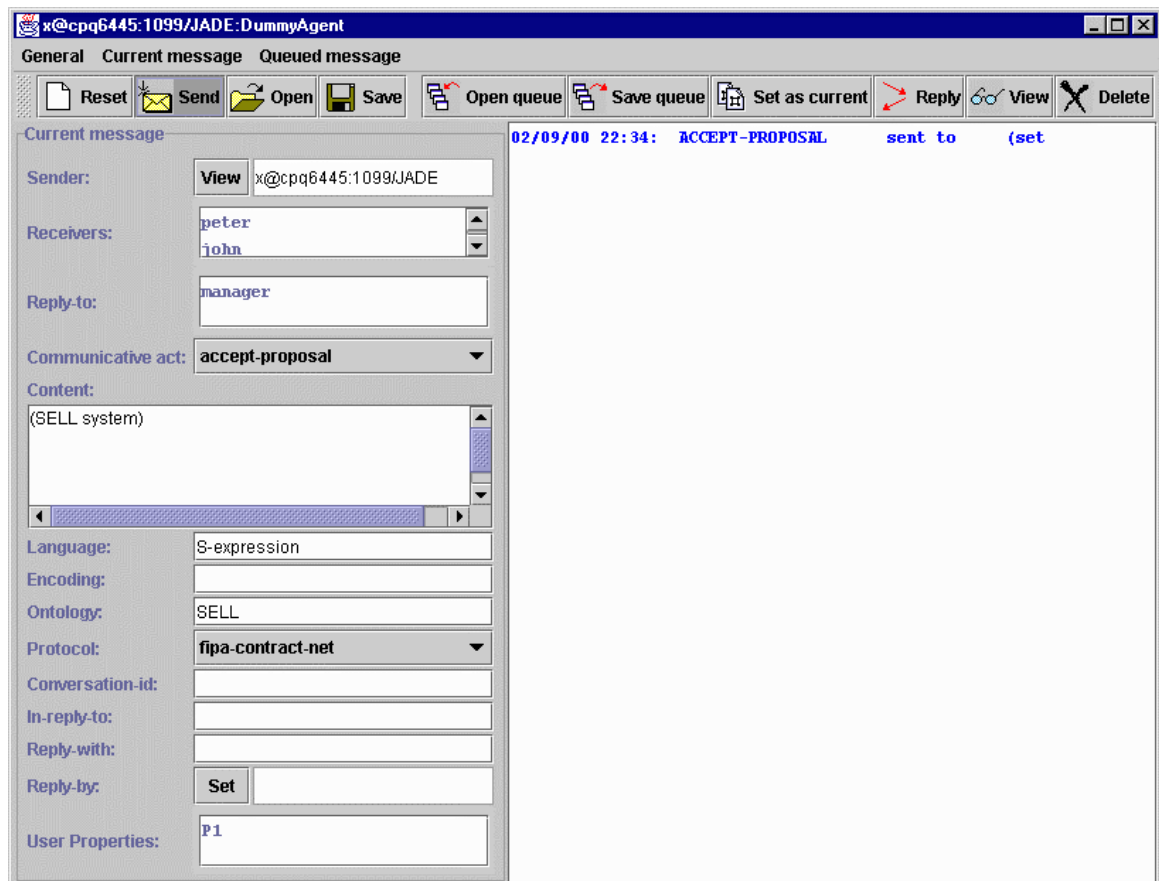


Figure 10 - Snapshot of the DummyAgent GUI

### 6.3 DF GUI

A GUI of the DF can be launched from the Tools menu of the RMA. This action is actually implemented by sending an ACL message to the DF asking it to show its GUI. Therefore, the GUI can just be shown on the host where the platform (main-container) was executed.

By using this GUI, the user can interact with the DF: view the descriptions of the registered agents, register and deregister agents, modify the description of registered agent, and also search for agent descriptions.

The GUI allows also to federate the DF with other DF's and create a complex network of domains and sub-domains of yellow pages. Any federated DF, even if resident on a remote non-JADE agent platform, can also be controlled by the same GUI and the same basic operations (view/register/deregister/modify/search) can be executed on the remote DF.

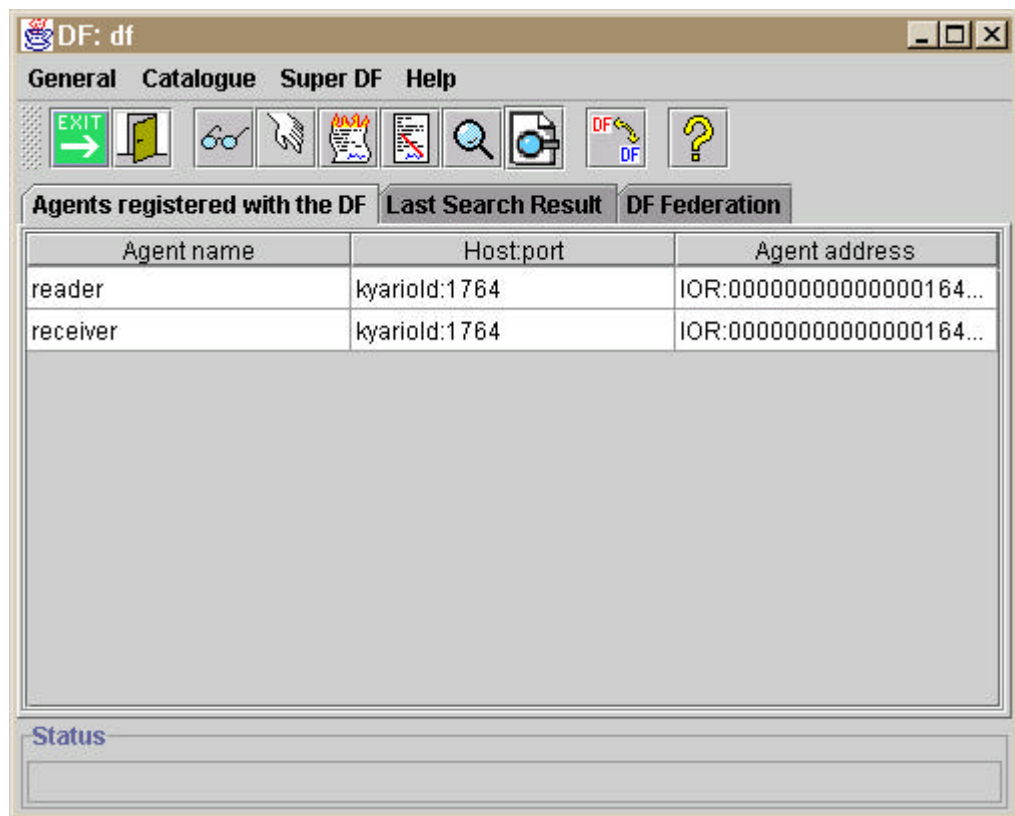


Figure 11 – Snapshot of the GUI of the DF

#### 6.4 Sniffer Agent

As the name itself points out, the Sniffer Agent is basically a Fipa-compliant Agent with sniffing features.

When the user decides to sniff an agent or a group of agents, every message directed to/from that agent / agentgroup is tracked and displayed in the sniffer Gui. The user can view every message and save it to disk. The user can also save all the tracked messages and reload it from a single file for later analysis.

This agent can be started both from the Tools menu of the RMA and also from the command line as follows:

```
Java jade.Boot sniffer:jade.tools.sniffer.Sniffer
```

The figure shows a snapshot of the GUI.

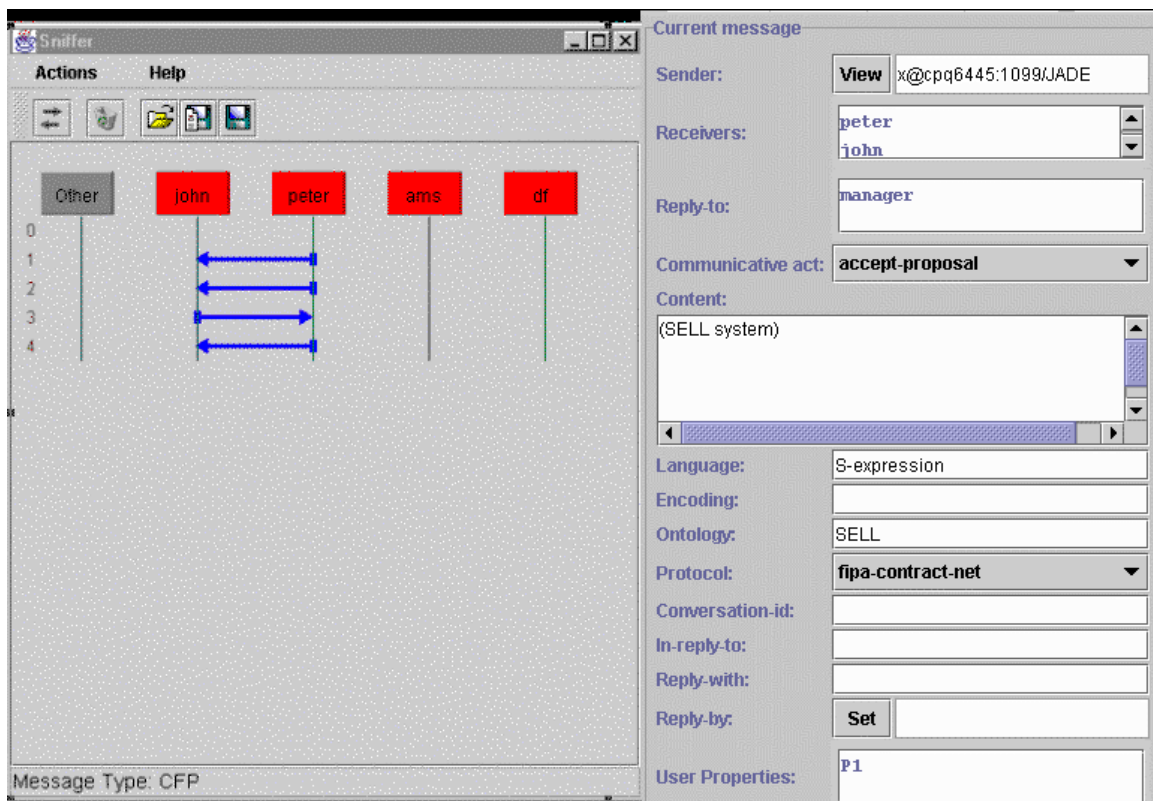


Figure 12 - Snapshot of the sniffer agent GUI

## 7 RELEASE NOTES

### 7.1 Major changes in JADE 2.01

- fixed bug about JADE integration with JSP (both the method Agent.doStart() and the examples). An inner class (SecurityManager) has been included within the Agent class as a first step towards managing security in JADE.
- fixed bug about jade.core.AgentManager inner interface that some JVM require to be declared public
- fixed bug in the sniffer that did not sniff agents on remote containers
- improved the configuration gui with help button and agent specifiers.

### 7.2 Major changes in JADE 2.0<sup>7</sup>

#### 7.2.1 Major changes

JADE 2.0 is based on and complies with the new FIPA 2000 specifications. The major changes in these specifications that affect JADE 2.0 are the following:

<sup>7</sup> JADE 2.0, and in particular the upgrade to FIPA 2000 specifications, has been undertaken in the context of, and partly financially supported by, the European project LEAP IST no. 1999-10211, that we gratefully acknowledge.

- the agent name is no more a String composed of the concatenation of the agent local name + the address of its home agent platform. It is now, instead, a structure, called AgentIdentifier or AID, composed of several slots: the globally unique identifier of the agent, its set of addresses, and its set of resolvers (i.e. the AMS and all those white-page agents where the agent is registered).
- the parameter envelope in the ACLMessage has been removed and two new parameters, reply-to and encoding, have been added. Also a couple of new performatives has been added.
- When a message is transmitted, an envelope is attached that contains information relevant to the delivery of the message. As a consequence, the IDL IIOP interface is also changed.
- The SL-0 parser has been improved and some bugs fixed. The concepts of set and sequence of terms have been introduced. The content of an ACLMessage must now be a tuple of expressions.
- The FIPA-Agent-Management ontology changed considerably and the AMS now supports a new action (search).

As a consequence of these changes in FIPA, and also based on the feedback received by the JADE users, the following is the list, probably non-exhaustive, of the major changes in JADE 2.0.

- The agent name is no more a String but an object of the class `jade.core.AID`
- The class `Agent` has been modified to work with AID's rather than Strings and by adding the two methods `getDefaultDF` and `getAMS()`
- All the methods to access the services of the DF and the AMS have been moved from the class `Agent` into the classes `DFServiceCommunicator` and `AMSServiceCommunicator`, that have been added to the package `jade.domain`.
- the class `ACLMessage` has been modified to comply with the new FIPA specifications and, as a consequence, also the `MessageTemplate` class has been updated. The methods `setSource` and `addDest` have been renamed into `setSender` and `addReceiver()`. The methods `dump()`, `toText()` and `fromText()` have been removed from this class. Care must be taken because the `getXXX()` methods return null, while in the previous release they returned an empty string, when the parameter has not been set.
- the interface `jade.lang.acl.Codec` has been added and the class `StringACLCodec` has been implemented to support String encoding.
- The interface of the content language codec (`jade.lang.Codec`) has been changed to take into account t-uples as a content of the `ACLMessage`.
- The IIOP IDL interface has been modified to comply with the new FIPA specs
- The names of some classes of the `jade.core` package have been improved and the `AgentGroup` class has been removed
- The class `FIPAAgentManagement` and its inner classes have been moved into the package `jade.domain.FIPAAgentManagement`
- A new package `jade.domain.JADEAgentManagement` has been added to include the JADE non-standard extensions to the FIPA-Agent-Management ontology.
- The support to user-defined ontologies has been modified by fixing the bug reported by Kaveh Kamyab, renaming some classes, improving the API and implementing the support to set, sequences, and unspecified types.
- The interface of the class `FIPARequestResponderBehaviour` has been slightly modified.
- A Message Transport Protocol (MTP) interface has been added to allow the future implementation of plug-and-play of new MTP's

- Some new classes have been added in the package `jade.gui` to display common concepts like an AID, a DFDescription, a ServiceDescription, ...
- the GUI of the DF has been modified to comply with the new FIPA-Agent-Management ontology and has been improved to allow the full control of a network of federated DF's. The classes implementing this GUI have been moved from `jade.gui` into `jade.tools.DFGUI`.
- the command line option `-conf` has been added and the option `-platform` has been set to default
- several sections of the programmer's guide have been rewritten in the hope of improving the clarity
- a new example was included about the integration of JADE within the JSP environment. The example, and a brief tutorial about that, was kindly provided by Daniel Le Berre that we gratefully acknowledge.
- The MeetingScheduler example has been rewritten in order to use the new support for user-defined content languages and ontologies and the bug reported by Matthew Mishrikey has been fixed.
- A bug, reported by Ijsbrand Zeelte, on agent cloning has been fixed; and a button to clone agents has been added to the MobileAgent example.

### 7.2.2 Guide to fast upgrade of the user code to JADE 2.0 and FIPA2000 specifications

Compliance to FIPA 2000 specifications required also a slight modification of the API's of JADE 2.0. At a first glance, the user might be discouraged in porting his code from JADE 1.X to JADE 2.0 because of the high number of errors that the JAVA compiler would report; however in all cases, it is just a question of automatic replacement of code. This section tries to help in this upgrading phase.

The followings are the most considerable modifications in FIPA and how they impact your user code:

- Command line options
  - The `-platform` option has been now set to the default. Therefore the command "java jade.Boot" does not start any more an agent container, instead it launches an agent platform. This default behavior is more intuitive than the old one.
- Agent class
  - According to the new FIPA specifications, the agent name, that was previously a String, has been replaced by an Agent Identifier, represented by the class `jade.core.AID` and composed of several attributes (i.e. name, addresses, resolvers). In particular the name attribute is the globally unique identifier of the agent and, conceptually, replaces the agent name of the previous version of JADE. The previous methods `Agent.getLocalName()` and `Agent.getName()` continue to exist and work at the same way; however the full name of an agent is no more composed of the concatenation of its local name and the IIOP address of the platform. On the other hand, FIPA specifications do not allow any more to decompose an agent name and distinguish between the local name and its home agent platform (HAP) address (i.e. the separator '@' is no more used). Further to these methods, a new method has been added in JADE, `Agent.getAID()`, that returns the Agent Identifier of the agent.

The constructor `AID(String guid)` is available to construct an Agent Identifier on the basis of the known string representing the name of the agent.

- All the methods to access the services of the DF and the AMS have been moved from this class into the classes `DFServiceCommunicator` and `AMSServiceCommunicator`, part of the package `jade.domain`. Therefore, it is necessary to replace any call to `Agent.registerWithDF(...)` with `DFServiceCommunicator.register(...)`
- The two methods `getDefaultDF()` and `getAMS()` have been added that return the AID of the default DF and of the AMS of the platform.
- Because the SL-0 language has been modified to represent t-uples, the two methods `fillContent` and `extractContent` require/return a `List` argument, where the `List` represents the t-uple. Each element of the t-uple is an Expression of the content language.
- ACLMessage class
  - the new FIPA specifications have added two new parameters `reply-to` and `encoding` to the ACL Message structure, the first representing the AID of the agents to which the reply message should be sent, the second representing the encoding scheme of the content. The specs also removed the `envelope` parameter that is now treated separately. The `sender` and `receiver` parameters do not contain any more `String`'s, but `AID`'s. As a consequence of that, the user code must be modified as follows:
    - replace `getSource()` and `setSource(String)` with `getSender()` and `setSender(myAgent.getAID())`
    - replace `getDest`, `getFirstDest()`, `addDest()` with `getAllReceiver()` (that returns an `Iterator` over all the receivers), `clearAllReceiver()`, `addReceiver(AID)`, `removeReceiver(AID)`.
    - replace `setReplyTo` and `getReplyTo` with `setInReplyTo` and `getInReplyTo`. This modification has been made to avoid ambiguity with the new `reply-to` parameter of FIPA.
    - replace `setEnvelope(String)` and `getEnvelope(String)` with `addUserDefinedParameter("X-envelope", String)`, `getUserDefinedParameter("X-envelope")` because `envelope` is no more a parameter of an `ACLMessage` in the FIPA specs.
    - take care of replying to messages by using the `AID`'s in the `reply-to` parameter, when present, instead of those in the `sender` parameter. The best practice is to use the method `ACLMessage.createReply()` that already takes care of that.
    - Care must be taken because the `getXXX()` methods return `null`, while in the previous release they returned an empty string, when the parameter has not been set.
    - the deprecated method `dump()` has been removed, replace it with `System.out.println(msg.toString())`.
    - the deprecated method `setType(String)` has been removed, replace it with `setPerformative(ACLMessage.REQUEST)`.



- the methods `fromText()` and `toText()` have been removed and replaced by the methods `decode` and `encode` in `jade.lang.acl.StringACLCodec`
- class `MessageTemplate`
  - this class has been updated by adding methods to match the two new parameters `encoding` and `reply-to` and by removing the method to match the envelope
  - just remind to rename `MatchReplyTo` with `MatchInReplyTo`
- class `AgentGroup`
  - this class has been removed because obsolete. No class has been provided to replace this one: infact, a group of agent identifiers can easily be represented by a `java.util.ArrayList()`
- class `jade.domain.FIPAAgentManagement` and its inner classes
  - this class in JADE 1.4 included a number of inner classes to represent the concepts of the FIPA-Agent-Management ontology, included concepts like `DFAgentDescription`, `ServiceAgentDescription`, ...  
All this classes have been updated to the new FIPA specifications and have been moved into the package `jade.domain.FIPAAgentManagement`. There is now one class for each concept of the ontology. Every class is a simple collection of the attributes defined by FIPA, with public methods to read and write them, as described in section 3.1.1
- class `jade.lang.SL0Codec` and interface `jade.lang.Codec`
  - because FIPA specifications have modified the SL-0 content language to allow only t-uples as content, the `encode` and `decode` methods have been modified to work now with `List` rather than `Object`.
- Package `jade.onto` and support to user-defined ontologies
  - This support has been simplified and improved by fixing a bug and adding support for set, sequences, and values of undefined type. Its quality, unfortunately, is not yet satisfactory because it suffers from a lack of standardization from FIPA of an abstract content language and an abstract ontology. Therefore, it might be modified again in the future. Refer to section **Error! Reference source not found.** for a detailed description of its usage. The main modifications are the following:
    - The class `TermDescriptor` has been renamed into `SlotDescriptor`.  
The constructor of a `SlotDescriptor` has 4 arguments. The first is the name of the slot and can be omitted for unnamed slots. The second is the category of the slot (i.e. `FRAME`, `SET`, `SEQUENCE`, `PRIMITIVE`, `ANY`). The third identifies the type of the values of the slot (e.g. `STRING_TYPE`, `LONG_TYPE`, ...). The last indicates if the presence of the slot is mandatory or optional.
    - The class `RoleFactory` has been renamed into `RoleEntityFactory`
    - The class `jade.onto.basic.BasicOntology` represents an ontology with basic concepts, like `Action`, `TruePredicate`, `AID`, ... It can be now joined to any user-defined ontology.
- Class `FIPAResponderBehaviour`
  - The interface of this class and of its inner classes (i.e. `ActionHandler` and `Factory`) has been modified.

- The method `create()` of the `Factory` has now an argument which is the received request `ACLMessage`
- The inner class `Action` has been renamed into `ActionHandler` in order to avoid ambiguity with `jade.onto.basic.Action`.
  - A second argument, that is the received request `ACLMessage`, has been added to its constructor.
  - All the methods `sendRefuse`, `SendAgree`, `SendFailure`, `SendInform`, `SendNotUnderstood` have been joined into a single method `sendReply` with two arguments: the performative and the content of the message.
- A new method `getActionName` has been added to the `FIPAResponseResponderBehaviour` class. The default implementation works only if both the content language and the ontology of the received request message were registered with the agent; otherwise it throws an exception. In general, it is expected that users have to override this method with their application-dependent implementation.

### 7.3 Major changes in JADE 1.4

- The visibility of the two methods `getContentBase64` and `setContentBase64` in `ACLMessage` has been restricted to private. They have been replaced by `setContentObject` and `getContentObject` as kindly suggested by Vasu Chandrasekhara (EML). This is the only modification that might impact the source code of your application agents.
- Agent mobility, including the migration of the code, has been implemented.
- Support to user-defined content languages and ontologies has been implemented. In this way, the encoding of the content message (e.g. `String`) is completely hidden to the programmer that can internally use Java objects instead.
- A bug has been removed that caused an agent deadlock when waiting for messages.
- The GUI of the Directory Facilitator has been completed, including the possibility of expressing constraints to the search operation and creating a federation of DFs.
- The death of a container is now notified to the RMA and the GUI is automatically updated.
- All the examples have been improved and more examples have been added.
- The method `MatchType` in `MessageTemplate` has been deprecated and replaced by `MatchPerformative`
- All deprecated calls have been removed, just the call to the methods, not the methods themselves. Notice that we plan to remove all deprecated methods in the next release of JADE.
- Added "About" in all the GUIs.
- More than one RMA can now be executed on the same container
- The internal representation of the performative in the class `ACLMessage` is now an integer and no more a `String`
- From the command line it is no more possible to pass both `"-platform"` and `"-host"` parameters.

### 7.4 Major changes in JADE 1.3

- Made JADE Open Source under LGPL License restrictions.

- Removed some bugs.
- Ported the GUI of the DF in Swing.
- Added some examples.

### **7.5 Major changes in JADE 1.2**

- Sniffer Agent. From the main GUI you can run the so-called sniffer agent that allows you to sniff and log the messages sent between agents.
- ACLMessage class. We have improved this class by deprecating the usage of Strings when you set/get the performative of a message (i.e. "request", "inform", ...). As probably you have noticed already, the usage of Strings requires you to remember using case insensitive comparison. Now a set of constants has been defined in the ACLMessage class: ACLMessage.INFORM, ...
- removed some bugs
- improved the documentation.

### **7.6 Major changes in JADE 1.1**

- support for Java serialization and transmission of sequence of bytes
- removed a bug in the DF parser that did not allow the registration of attribute values starting with a ':' character
- introduced support for intra-platform mobility of agents. This feature is still at an experimental level, testing is still on-going and no documentation is yet available.
- made case-sensitive the class jade.core.AgentGroup and the agent names in the ACLMessage class
- introduced support for Fipa-Iterated-Contract-Net protocol.

### **7.7 Major changes in JADE 1.0**

- Timeout support on message receive, both agent-level and behaviour-level. Now a timeout version of blockingReceive() is available, and a new constructor has been added to ReceiverBehaviour class to support timeout. Moreover, a block() version with timeout has been added to Behaviour class.
- AgentReceiver example program in directory src/examples/ex2 was modified and now times out every 10 seconds if no message is received.
- A new example program, AgentTimeout, was added to directory src/examples/ex2 to show new ReceiverBehaviour support for timeouts.
- Moved all behaviours in a separate package. Now they are in jade.core.behaviours package. User application must be updated to either 'import jade.core.behaviours.\*' or the individual behaviours they use.
- Moved RMA agent in a different package; now the Remote Management Agent has class named jade.tools.rma.rma. It can still be started with '-gui' command line option.
- Multiple RMAs can be started on the same platform; they can be given any name and still perform their functions.
- When requesting AMS actions via an RMA, if a 'refuse' or a 'failure' message is received, a suitable error dialog pops out, allowing to view the ACL message.
- Fixed a bug in RMA graphical user interface, which sometimes caused a deadlock on platform startup.

- When an ACL message has an empty 'sender' slot, JADE puts the sender agent name in it automatically.
- Added a new tool agent: `jade.tools.DummyAgent.DummyAgent`, allowing ACL messages manipulations through a GUI. This agent can be either started from an RMA or directly, since is an ordinary agent.
- Javadoc-generated HTML documentation for all JADE public classes.
- `jade.domain.FipaRequestClientBehaviour` has been replaced by `jade.proto.FipaRequestInitiatorBehaviour` and a new `FipaRequestResponderBehaviour` has been added in `jade.proto` package.
- ACL messages now can have many agent names in 'receiver' slot. This required making some changes to `ACLMessage` interface. Please see HTML documentation for `ACLMessage` class.
- Agents can be suspended and resumed by the GUI or they can suspend themselves; user should not suspend neither the AMS nor the RMA, since they obviously wouldn't be able to resume then back.
- Now an agent can create another agent and starting it up with `Agent.doStart(String name)`.
- The gui of the DF is not shown at startup of the platform but only when requested via the RMA gui.

## 7.8 Major changes in Jade 0.97

- IIOP support for inter-platform communication (see section 5.2).
- Now JDK 1.2 is required to run JADE.
- GUI for DF management.
- Modified Agent class (`getName()/getLocalName()` methods). (may need change to your source file!)
- Modified AgentGroup class. (may need change to your source file!)
- Renamed `ComplexBehaviour.addBehaviour()` to `addSubBehaviour()`. (may need change to your source file!)
- Renamed `ComplexBehaviour.removeBehaviour()` to `removeSubBehaviour()`. (may need change to your source file!)
- Implemented some FIPA interaction protocols.
- Added javadoc documentation.

## 7.9 Major changes in Jade 0.92

- Complete support for FIPA system agents.
- Many examples revised, particularly `examples.ex5.dfTester` now is complete and usable.
- Added `examples.ex5.subDF` example to show multiple agent domains.
- Added `examples.ex2.filebasedSenderAgent` to help in debugging and testing.
- Updated and completed documentation, which is now included as part of JADE package.

### 7.10 Major changes in Jade 0.9

- RMI Registry included within the Agent Platform. Therefore, it is no more necessary to run a shell with the rmiregistry;
- added reset() method in the Behaviour class to allow complex and repetitive behaviours;
- modifications to the method in the Agent class that make the state transitions: doDelete and doWake() to wake-up all the blocked behaviours. Therefore to wake-up an agent it is possible to call the method doWake() or to send it a message;
- included the RMA (remote management agent), that is a GUI to control the agent platform. This agent can be executed via the option '-gui' on the command line or as a normal agent;
- the AMS is now able to create-agent, kill-agent, and manages subscribe messages from the RMA, and it is also able to manage several RMAs. When a new container or a new agent is born or is killed, the AMS sends an inform to all the registered RMAs;
- included option '-version' on the command line to print the current version of Jade.

### 7.11 Major changes from JADE 0.71+ to JADE 0.79+

- included this programmer's guide;
- implemented DF, AMS, and ACC that now are activated at the bootstrap of the Agent platform;
- modified Behaviour that is now an Abstract class and no more an Interface;
- included SenderBehaviour and ReceiverBehaviour that are more suitable to send and receive messages;
- implemented new Behaviours, like OneShotBehaviour, CyclicBehaviour;
- unified method name: the execute() method of the behaviours is now always called action();
- included an example, which is a ready-to-use agent, of Jess-based agent.