JADE PROGRAMMER'S GUIDE

last update: 5-June-2000.

TABLE OF CONTENTS

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 standard 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 guide.

JADE is written in Java language and is made by 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 includes four main packages.

`jade.core` contains the kernel of the system. It owns the `Agent` class with base agent features that must be extended by application programmers; besides, a `Behaviour` class hierarchy is contained in `jade.core.behaviours` sub-package. Behaviours are logical activity units for an agent and they can be composed in various ways to achieve complex execution patterns. 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.

The `jade.domain` package contains all the Java classes to represent FIPA agent platform and domain models, such as standard Agent Management entities, languages and ontologies (e.g. the mandatory AMS, DF and ACC agents).

Finally, `jade.proto` is the package that contains classes to model FIPA standard interaction protocols (i.e. '*fipa-request*', '*fipa-query*' and so on), as well as classes to help application programmers to create protocols of their own.

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

- A *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, but then more than one GUI can be activate. JADE maintains coherence among multiple RMAs by simply multicasting changes to all of them. Moreover, RMA console is able to start other JADE tools.
- A *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; another part of the dummy agent is able to display in a list all ACL messages sent or received, complete with timestamp information in order to allowing agent conversation recording and rehearsal.

- A *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.
- A *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. This agent is useful, e.g. to handle network firewalls or to provide platform interactions with Java applets within a web browser.

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:

- 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;
- Distributed agent platform. The agent platform can be split among several hosts (provided that RMI connections are possible between them). 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 runtime support to them. Java events are used for communication between agents on the same host. Concurrent tasks can be still executed by one agent, and JADE schedules these tasks in a non preemptive fashion;
- 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.
- FIPA97-compliant IIOP protocol to connect different agent platforms;
- Efficient transport of ACL messages inside the same agent platform, as 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 string format. The 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;
- Graphical user interface to manage several agents and agent containers from the same agent;
- Debugging tools to help in developing multi agents applications based on JADE;
- Support to the usage of application-defined content languages and ontologies;
- Intra-platform agent mobility, including state and code.

3. CREATING 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 FIPA97 specifications.

3.1 Using the Agent class

`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 must 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.

A JADE agent can be in one of several states, according to Agent Platform Life Cycle in FIPA 97 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 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.

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 97 specification, Part 1 Section 7.6, Figure 3. For example, `doWait()` method puts the agent into AP_WAITING state from AP_ACTIVE state.

An agent executes its behaviours only in AP_ACTIVE state, so if some behaviour calls the `doWait()` method the whole agent is blocked and not just the calling behaviour. A `block()` method is provided in `Behaviour` class to suspend a single agent behaviour (see section 3.2 for details).

The framework requires that the programmer overrides `setup()` and `takeDown()` methods in order to insert his/her own initialisation and cleanup functions into the agent. When `setup()` method starts, 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) set the delegate-agent and forward-address, if necessary, and modify the data registered with the AMS (see section 3.3);
- (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().` They are the actual behaviours of the agent and are scheduled immediately after the end of the `setup()` method;

For a correct implementation, at least one behaviour must be added within the method `setup()`. At the end of the method `setup()`, JADE automatically passes to execute the first behaviour found in the queue of ready tasks. `Agent` class has a couple of methods: `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)`, to manage the tasks ready queue of a specific agent.

The `takeDown()` method is executed when the agent is about to go to AP_DELETED state; the agent is still registered with the AMS and can therefore send messages to other agents. The intended purpose of this method is to perform application specific cleanup operations, such as deregistering with DF agents.

In order to realise the communication among agents, the framework provides the agent developer with low-level primitives embedding inter-agent communication mechanisms, and with agent interaction protocols, implemented as customisable behaviours. Objects of the `ACLMessage` class must be used to send and receive messages (see section 3.5).

The `send()` method allows to send an `ACLMessage` to an agent or to an agent group (notice that in this case the method directly sets the receiver parameter to, in turn, the name of each receiving agent). The method call is completely transparent to where the agent resides, i.e. be it local or remote; it is committed to the container internal communication mechanisms. The value of `ACLMessage :receiver` slot indicates the list of the receiving agent names, according to the Fipa97 specifications. When using concurrent behaviours, the `SenderBehaviour` can be used instead of this method call. In fact, this behaviour yields the control, allowing the other behaviours to be scheduled.

Two variants of the `receive()` method are available for receiving messages. The first returns the first available ACL message object in the agent message queue. The latter, instead, returns the first message matching the pattern specified in the method call. Both methods are non-blocking, returning `null` when no suitable messages are available. A blocking version of the same two variants is also available, but much care must be taken as it ***causes the suspension of the complete agent activity and in particular of all its Behaviours***. Therefore, when using concurrent behaviours, better performance can be obtained by adding an instance `ReceiverBehaviour` to a `SequentialBehaviour` instead of this method calls. In fact, this is a behaviour that yields the control and allows the other behaviours to be scheduled.

3.1.1 AgentGroup class

In order to help the developer to send multicast messages, JADE framework defines an `AgentGroup` class. The functionality of such class is to hold a set of agent names, seen as a single group, and to provide means for iterating along the names. Special versions of the `send()` method are provided, taking an `AgentGroup` as argument; they send the given ACL message to every agent in the group.

3.1.2 MessageTemplate class

According to FIPA specifications, an agent must be able to carry on many conversations simultaneously; special ACL message fields can be used to aid in distinguishing between different ongoing conversations (e.g. `:conversation-id` and `:reply-with` fields) .

Therefore, it is often useful to search for the first ACL message in the queue with specific values for one or many of its fields. The `MessageTemplate` class allows to build patterns to match ACL messages against. Elementary patterns can be combined with AND, OR and NOT operators, in order to build more complex matching rules.

3.2 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 contest 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 ad 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.

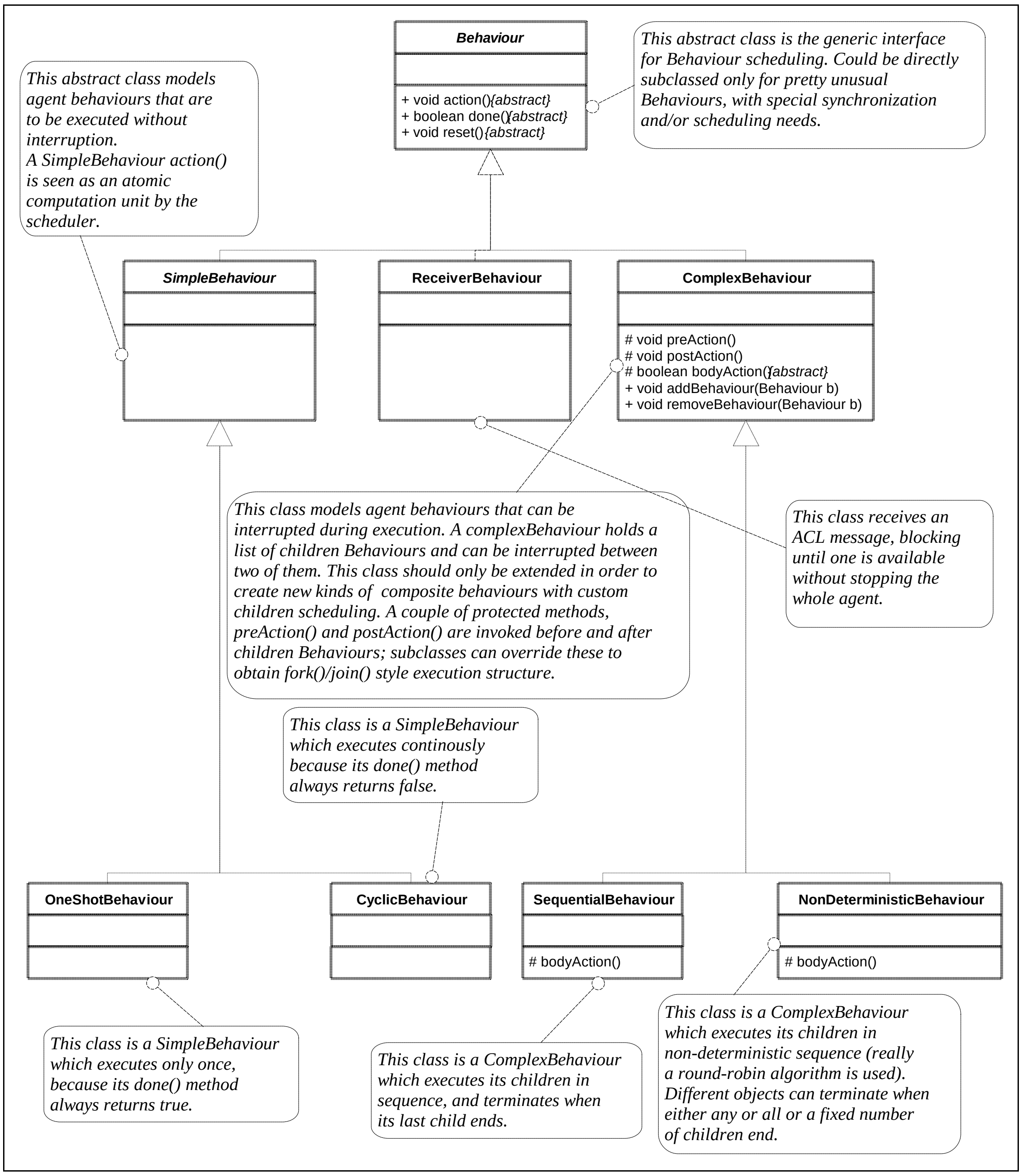

Behaviour
+ void action(){abstract}
+ boolean done(){abstract}
+ void reset(){abstract}
This abstract class is the generic interface for Behaviour scheduling. Could be directly subclassed only for pretty unusual Behaviours, with special synchronization and/or scheduling needs.
This abstract class models agent behaviours that are to be executed without interruption. A SimpleBehaviour action() is seen as an atomic computation unit by the scheduler.
SimpleBehaviour
ReceiverBehaviour
ComplexBehaviour
# void preAction()
# void postAction()
# boolean bodyAction(){abstract}
+ void addBehaviour(Behaviour b)
+ void removeBehaviour(Behaviour b)
This class models agent behaviours that can be interrupted during execution. A complexBehaviour holds a list of children Behaviours and can be interrupted between two of them. This class should only be extended in order to create new kinds of composite behaviours with custom children scheduling. A couple of protected methods, preAction() and postAction() are invoked before and after children Behaviours; subclasses can override these to obtain fork()/join() style execution structure.
This class receives an ACL message, blocking until one is available without stopping the whole agent.
This class is a SimpleBehaviour which executes continously because its done() method always returns false.
OneShotBehaviour
CyclicBehaviour
SequentialBehaviour
# bodyAction()
NonDeterministicBehaviour
# bodyAction()
This class is a SimpleBehaviour which executes only once, because its done() method always returns true.
This class is a ComplexBehaviour which executes its children in sequence, and terminates when its last child ends.
This class is a ComplexBehaviour which executes its children in non-deterministic sequence (really a round-robin algorithm is used). Different objects can terminate when either any or all or a fixed number of children end.

Figure A - 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 "true" 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 `ComplexBehaviour`s 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.

3.2.1 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.

3.2.2 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.

3.2.3 class OneShotBehaviour

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

3.2.4 class CyclicBehaviour

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

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.2.8 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.

3.2.9 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.

3.2.10 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 it self.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.StringReader;
import java.io.OutputStreamWriter;
import java.io.BufferedWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InterruptedIOException;
import java.io.IOException;

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.setSource(getLocalName());
            msg.addDest(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.BufferedWriter;
import java.io.OutputStreamWriter;
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 m1andm2_and_notm3 =
MessageTemplate.and(m1andm2, 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(m1andm2_and_notm3);
      if (msg!= null){
        System.out.println("\nAgent "+ getLocalName() + " received
the following message in state 1.1: ");
        msg.toText(new BufferedWriter(new
OutputStreamWriter(System.out)));
        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.toText(new BufferedWriter(new
OutputStreamWriter(System.out)));
      }
      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 m1andm2 = MessageTemplate.and(m1,m2);

      MessageTemplate m1andm2_and_m3 =
MessageTemplate.and(m1andm2, m3);
      //blockingReceive and template
      ACLMessage msg = blockingReceive(m1andm2_and_m3);

      if (msg!= null){
        System.out.println("\nAgent "+ getLocalName() + " received
the following message in state 1.3: ");
        msg.toText(new BufferedWriter(new
OutputStreamWriter(System.out)));
      }
      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);

  }


}
```

Figure 1 – Java code of two sample agents

3.3 Using FIPA System Agents

FIPA 97 standard mandates the presence of AMS, DF and ACC agents on every compliant agent platform; so JADE has the three of them running as soon as a platform is started. JADE system agents use `SL0` language and `fipa-request` interaction protocol to communicate with application agents and `fipa-agent-management` ontology as the framework for conceptual agreement between the communication agents. The ontology used by JADE system agents conforms to the specification given in FIPA 97 standard document, apart from ontology grammar that follows FIPA 98 0.1 standard draft (Part 1 Section 10.1) because it was broken in FIPA 97 version 1.0.

All the mandatory elements of a FIPA compliant Agent Platform are implemented by Java classes of `jade.domain` package; particularly, `fipa-agent-management` ontology is realised by `AgentManagementOntology` Java class. This class contains inner classes that model the various ontology elements.

In order to ease application programming, all `AgentManagementOntology` inner classes are kept very simple and share a common design, so that the same coding idioms can be adopted for any ontological entity.

Since ACL message content is a raw string, all `AgentManagementOntology` inner classes have a pair of methods to perform conversions to/from character stream objects: a `fromText(Reader r)` static factory method builds a new ontology object out of the content of a `java.io.Reader` object, whereas `toText(Writer w)` method writes an existing object onto a suitable `java.io.Writer` object. These two methods work just like *Java Serialization API*, letting programmers deal with objects all the time and converting them to an external format only for storage or transmission.

Every inner class 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; if the class has an attribute named `attr` of type `attrType`, two cases are possible:

1) The attribute has a single value; then it can be read with `attrType getAttr()` and written with `void setAttr(attrType a)`; every call to `setAttr()` overwrites any previous value of the attribute.
2) The attribute has a list of values; then there is an `void addAttr(attrType a)` method to insert a new value and a `void removeAttrs()` method to remove all the values (the list becomes empty). Reading is performed by a `Enumeration getAttrs()` method; then the programmer must walk the `Enumeration` and cast its elements to the appropriate type.

A list of these inner classes follows:

3.3.1 AMSAgentDescriptor class

This class models `FIPA-AMS-description` ontology object and has the following attributes:

- String name
- String address
- String signature
- String APState
- String delegateAgentName
- String forwardAddress
- String ownership

3.3.2 DFAgentDescriptor class

This class models `FIPA-DF-description` ontology object and has the following attributes:

- String name;
- Vector addresses ;
- Vector services;
- String type;
- Vector interactionProtocols;
- String ontology;
- String ownership;
- String DFState;

3.3.3 ServiceDescriptor class

This class models `FIPA-Service-Desc-Item` ontology object and has the following attributes:

- String name;
- String type;
- String ontology;
- String fixedProperties;
- String negotiableProperties;
- String communicationProperties;

3.3.4 Constraint class

This class models `Constraint` ontology object and has the following attributes. The name of a constraint object can be set either to `Constraint.DFDEPTH` or to `Constraint.RESPREQ`; the function attribute must be set to `Constraint.MIN`, `Constraint.MAX` or `Constraint.EXACTLY`. The argument attribute must be a positive integer number.

- String name;
- String fn;
- int arg;

3.3.5 DFSearchResult class

This class models the set of `DFAgentDescriptor` objects returned as result by a `search` action of a DF agent. It behaves just like a `java.util.Hashtable` containing `DFAgentDescriptor` objects, indexed by agent names.

Since objects of this class are used for communication between a client agent and a DF agent, access methods can throw FIPA exceptions when called. The following is the list of the public methods of this class.

3.3.6 Action interface

This interface is implemented by various classes which represent actions performed by FIPA system agents, together with their arguments. Each one of these class has a `name` and an `actor` attributes: the `name` is the name of the action (e.g. `register-agent` or `search`), whereas `actor` is the name of the agent that is to perform the action.

Besides, each action class must have a `toText()` method to write itself onto a character stream, and a `fromText()` static Factory Method to be built out of a character stream. To send a message containing an action, one has to convert it to a string first, then set message content to the string representation of the action object: for example, willing to register itself with a DF, an agent can use the following java code:

```
ACLMessage requestMsg = ...; // Set fields
AgentManagementOntology.DFAgentDescriptor  dfd  =  ...;  //  Set
fields

AgentManagementOntology.DFAction a =
  new AgentManagementOntology.DFAction();
a.setName(AgentManagementOntology.DFAction.REGISTER);
a.setActor(“myDF”);
a.setArg(dfd);
StringWriter text = new StringWriter();
a.toText(text);
request.setContent(text.toString());
```

Every class implementing `Action` interface will add attributes and methods as needed by the specific action; in the code above, a `DFAction` class adds a `DFAgentDescriptor` argument, containing data to be stored in DF database. Following is the `Action` interface definition.

3.3.7 AMSAction class

This class can be used to model any one of `register-agent`, `deregister-agent`, `modify-agent`, or `authenticate` standard AMS actions, besides being used by administrative GUI for platform management operations. It takes a single argument of `AMSAgentDescriptor` class.

3.3.8 DFAction class

This class can be used to model any one of `register`, `deregister`, or `modify` standard DF actions. It takes a single argument of `DFAgentDescriptor` class.

3.3.9 DFSearchAction class

This class can be used to model `search` standard DF action. It takes an argument of `DFAgentDescriptor` class and a `Vector` of `Constraint` objects. This class lacks a specific `fromText()` method because `DFAction.fromText()` can recognise a `search` action and creates directly a `DFSearchAction` object.

3.3.10 ACCAction class

This class can be used to model `forward` standard ACC action. It takes a single argument of `ACLMessage` class.

3.4 Simplified API for System Agents access

JADE features described so far allow complete interactions between FIPA system agents and user defined agents; to successfully accomplish this task an application programmer has to:

- ✓ Build a Java object representing the message content (e.g. a `DFAction`).
- ✓ Convert the object to a string using `toText()` method.
- ✓ Create a `"request"` ACL message, set its content to the string representation of the object and send it to the system agent.
- ✓ Receive reply from the system agent, handling all possible cases of `"fipa-request"` protocol.
- ✓ Parse reply content, maybe building a Java object out of it.

This is the most general way to access FIPA platform features, but requires a certain amount of work by the programmer. For common, predefined interaction such as modifying AMS data of an agent or searching a DF for information, though, some predefined APIs are provided.

These APIs can greatly simplify programming, not only because they are ready to use, but also because they hide string based content parsing and error handling from the application: a JADE programmer can deal exclusively with Java objects for content representation and Java exceptions for error handling, without any type-unsafe string interfaces.

For each interaction, two access ways are provided: a method to call in `Agent` class and a behaviour to add, defined in `jade.domain` package. The main difference between the two is that `Agent` methods use `blockingReceive()` and stop the whole agent, whereas behaviours can engage in a conversation to arrive while still allowing their agent to do other tasks.

3.4.1 Agent class API

A different method is exported by `Agent` class for every system agent action; all these methods can throw a `jade.domain.FIPAException` when receiving `not-understood`, `failure` or `refuse` messages from their peer agent and take suitable `AgentManagementOntology` objects as parameters.

There are four methods in `Agent` class to access AMS agent features: `registerWithAMS()` and `deregisterWithAMS()` methods are called automatically by JADE just before calling `setup()` method and just after `takeDown()` method returns. So there is no need for the user to call them (however, calling `registerWithAMS()` a second time will result in a `jade.domain.AgentAlreadyRegisteredException` exception being thrown).

As of JADE 1.3, AMS authentication has not been implemented. Moreover, these four methods will be soon modified to take a single ontology object as parameter.

The `forwardWithACC()` method takes an ACL message and uses ACC agent to bounce it to one or more recipient agents.

There are also four methods to access DF agent features; application programmers must simply build a suitable `DFAgentDescriptor` object and pass it to the method; a `dfName` argument allows to select different DF agents, i.e. to work with multiple Agent Domains (to access default DF, the string `”df”` must be used). The three methods `registerWithDF()`, `deregisterWithDF()` and `modifyDFData()` work exactly the same way, engaging the agent in a blocking interaction protocol with the chosen DF.

Searching a DF for information works in a slightly different way. The programmer has to pass a `Vector` containing one or more `Constraint`s to `searchDF()` method; the search will be done using `dfd` parameter as a match template and `constraints` as search constraints (one can pass `null` to specify no constraints). A `DFSearchResult` is returned containing all `DFAgentDescriptor` objects matching `dfd` argument. In case of errors no exception is thrown, but the returned `DFSearchResult` is put in an invalid state that will cause it to throw an appropriate `FIPAException` as soon as it is accessed.

JADE DF agents try to detect inconsistencies within search constraints and send `refuse` or `failure` messages accordingly; these messages are then unmarshalled into Java exceptions and put in returned `DFSearchResult` object. Recursive searches can be performed using `”:df-depth”` search constraint; when a `df-depth` greater than one results from `”:df-depth”` constraints, all sub-DF registered with target DF are searched for other matches (using a `”:df-depth”` lesser by one), then all results are gathered and sent back to requesting agent. Since a `SearchDFBehaviour` is used, all the searches can be performed in parallel.

The “:resp-req” search constraint is still checked for consistency, but its final value is currently ignored.

3.4.2 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 (an agent starting the conversation) and the *Responder* role (an 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.

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

FipaRequestInitiatorBehaviour

The various actions exposed by JADE system agents all comply to standard `“fipa-request”` protocol, so that an agent willing to request any one of them must always engage itself in a `“fipa-request”` interaction.

JADE interaction protocols package provides both a `FipaRequestInitiatorBehaviour` and a `FipaRequestResponderBehaviour` class to do this; application programmers must extend one of the two according to the role their agent must play within a conversation and implement handler methods for various message kinds arising from a `“fipa-request”` interaction.

A `FipaRequestInitiatorBehaviour` object must be passed three arguments at construction time: the agent the behaviour belongs to, the ACL request message and the message template to match replies against.

The programmer must set `:receiver` and `:content` fields of the request message; JADE will set message type to `"request"` and `:protocol` field to `"fipa-request"`. The third constructor argument is a message template that will be used to match incoming replies; replies will have to match user defined template **and** have `"fipa-request"` as `:protocol` field **and** come from the correct agent. Besides, if `":conversation-id"` and `":reply-with"` fields are set, they will be used to match corresponding replies.

When a reply message is received from `"fipa-request"` 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.

SearchDFBehaviour

Using `FipaRequestInitiatorBehaviour` class, it is fairly easy to write behaviours for all FIPA system actions, but a subclass for `search` DF action is already available in `jade.domain` package; using `SearchDFBehaviour` class, an agent can access multiple DF agents concurrently and still perform other duties. The constructor takes five parameters:

i. The agent the behaviour belongs to.
ii. The ACL request message to send.
iii. The `DFAgentDescriptor` to use to search DF database.
iv. A `Vector` containing search constraints (or `null` for no constraints).
v. A `DFSearchResult`, to fill with `DFAgentDescriptor` objects retrieved from target DF. If something goes wrong with the DF, a `FIPAException` will be thrown by `DFSearchResult` object when result access is attempted.

The first two parameters are directly passed to `FipaRequestInitiatorBehaviour` base class constructor, along with a `MessageTemplate` object matching `"SL0"` as `:language` field and `"fipa-agent-management"` as `:ontology` field.

FipaRequestResponderBehaviour

documentation still to do.

FipaQueryInitiatorBehaviour

The `FipaQueryInitiatorBehaviour` implements the initiator role in `fipa-query` interaction protocol. By definition the protocol, and as a consequence the behaviour, terminates as soon as the last `inform` message is received.

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 FipaQueryInitiatorBehaviur:

- `public void handleOtherMessages(ACLMessage msg)` to handle all received messages that are different from "inform" message but that still have the right value of `:in-reply-to` parameter;

- `public void handleInformeMessages(Vector messages)` to handle the "inform" messages received in response to the query.

If present, this behaviour correctly uses the parameter `:reply-by` of the query message to set a timeout; if the timeout expires before any answer is received the method `handleInformeMessages` is execcuted by passing an empty vector of messages. By default, this timeout is set to 1 minute.

The constructor of this behaviour class takes 3 parameters

```
public FipaQueryInitiatorBehaviour(Agent a, ACLMessage msg, AgentGroup
group)
```

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.

It must be taken care to late answers that arrive after the timeout expires because this behaviour does not consume them.

FipaQueryResponderBehaviour

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

The abstract method `processQuery` 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.

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 message (call for proposal) to a set of agents. See the API javadoc documentation for an in-depth explanation of how to use this class. In that documentation is also explained how to use this same class in order to implement the ***FipaIteratedContractNet Protocol***

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. See the API javadoc documentation for an in-depth explanation of how to use this class.

3.5 The ACLMessage class

The class `ACLMessage` represents ACL messages that can be sent and received by an agent.

An agent willing to send a message should create a new `ACLMessage` object, fill its fields with appropriate values (using methods named `set<Parameter>()`), and call the method `send()` (implemented by the class `Agent`), or add a `SenderBehaviour` to the behaviour list.

Likewise, an agent willing to receive a message can call `receive()` or `blockingReceive()` (both in the `Agent` class), or add a `ReceiverBehaviour` to its behaviours. Received message fields can then be read using `get<Parameter>()` access methods.

Notice that these access methods never returns `null`. Non-initialized parameters are set to the empty string, instead.

Apart from get and set methods, `ACLMessage` has three more support methods: `reset()` resets the values of all message fields. `ToText()` writes the ACL message to a stream and a static Factory Method `fromText()` builds a new ACL message out of a character stream.

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.

3.5.2 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.

3.5.3 Support for Java serialization and transmitting sequence of bytes over an ACL Message

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()`. The HTML documentation of JADE API, reports an example of usage of this feature.

It must be noticed that serialized Java objects and, more in general, sequence of bytes should never be set to the content of an ACLMessage without an appropriate form of encoding. The FIPA inter-platform delivery mechanism, in fact, so far, gives no guarantee of intact delivery of the content. Moreover, JADE cannot recognize automatically the usage of Base64 encoding[1], so the methods must appropriately used by the programmers.

[1] 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.

3.6 Application-defined content languages and ontologies

The JADE framework provides support to the usage of content languages and ontologies designed for specific agent applications. This support is still minimal in nature, not yet fully settled, and further improvements and **modifications of the API are expected in the future releases**. However, its usage, testing and comments from the users are considered so much important to be still included in the official release of JADE. This section provides an overview of the implemented support, however the reader should refer to the javadoc documentation for an in-depth description of the available classes and methods.

The figure shows the implemented pipeline process. A content expression is encoded within an ACL message with a certain encoding (e.g. String) and content language. An appropriate content language *Codec* is able to parse the stream and convert it into a *Frame* object, that has a set of slots and slot values[2]. This Frame object is an internal representation of the content, which is independent of the chosen encoding, the content language, and the ontology. This internal representation is then fed into an appropriate *Ontology* object that is able to convert the frame into a *Concept* that belongs to the domain of discourse and that is implemented directly as a Java class. 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 to the programmer the stages of this pipeline as described in the following.

[2] Notice that the value of a slot might be another Frame object, as in the case of the following sentence *(User :name John :address (Address :street "Via Paolo Rossi" :city Canicattì))*

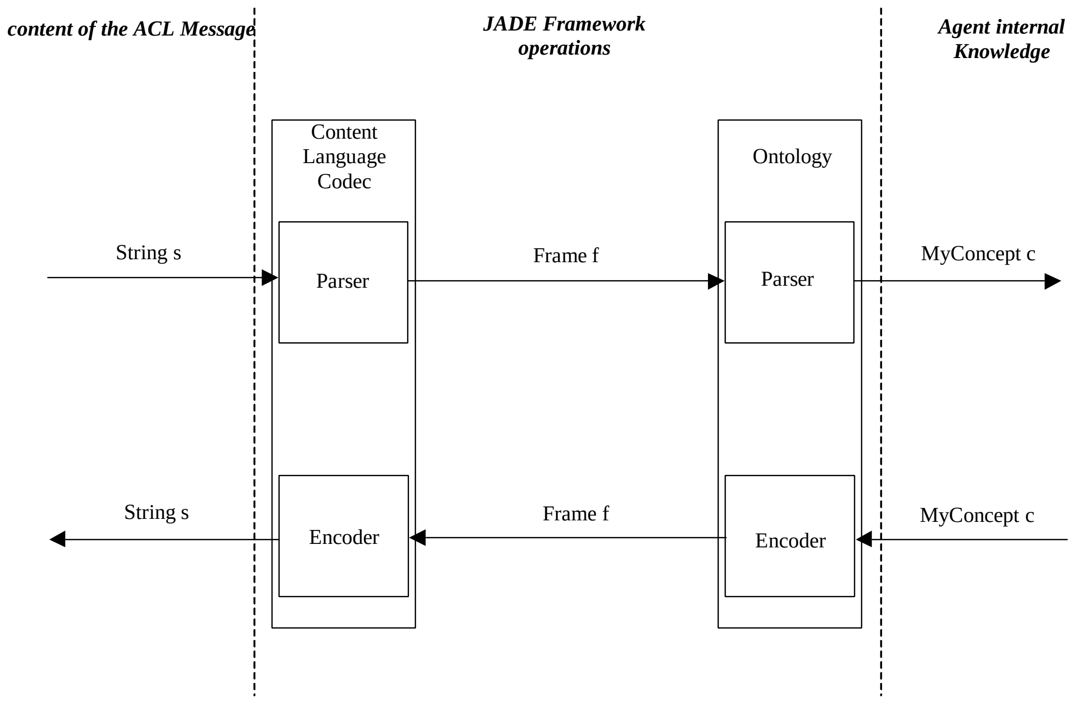


Figure 2 - Pipeline of the encoding of the message content

3.6.1 Codec of a Content Language

Each content language 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 into a Frame object, and encode the content from a Frame object into the content language syntax and encoding. Because the Frame class is a neutral type (i.e. it does not distinguish between actions, predicates, terms, functions, …), in some cases the codec operations may need the support of the ontology in order to distinguish between the possible types of frames. A set of methods is available in the *jade.onto.Ontology* class in order to support the codec operations. *Notice that it is under analysis the creation of a general framework of frame types independent of any specific content language, any suggestion from the users is welcome.* The Frame class 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 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.

Notice that JADE includes already the Codec for SL-0, that is the class *jade.lang.sl.SL0Codec*.

3.6.2 Creating an Ontology

In its most minimalistic interpretation, an ontology is a set of entities that compose the domain of discourse. In JADE, each entity can be represented as an application-dependent Java class (e.g. the User class) or as a generic frame by directly using the *Frame* class. It must be noticed that this is an operational view of the problem and it is not intended to have any theoretical support neither to replace, rather to complement, more authoritative approaches, like OKBC and Ontolingua.

Each ontology in JADE must implement the interface *jade.onto.Ontology*. A default implementation, *jade.onto.DefaultOntology*, is also provided that is simple but still expected to be useful in most practical applications. After constructing a new object of type *DefaultOntology*, the programmer must add to it the set of entities that compose the ontology and that will be later automatically used by the JADE framework.

```
DefaultOntology myOntology = new DefaultOntology();
```

Each entity must be described as an array of TermDescriptor, as in the following example.

```
myOntology.addFrame("Address", Ontology.CONCEPT_TYPE,
  new TermDescriptor[] {
    new TermDescriptor("STREET", Ontology.STRING_TYPE, Ontology.M),
    new TermDescriptor("CITY", Ontology.STRING_TYPE, Ontology.M)
                  });

myOntology.addFrame("Person", Ontology.CONCEPT_TYPE,
  new TermDescriptor[] {
    new TermDescriptor("NAME", Ontology.STRING_TYPE, Ontology.M),
    new TermDescriptor("AGE", Ontology.INTEGER_TYPE, Ontology.M),
    new TermDescriptor("ADDRESS", Ontology.CONCEPT_TYPE, "Address",

                Ontology.O)

                   });
```

Two frames, named "Address" and "Person", have been added to this sample ontology. The type of both is Concept, as opposed to Action and Predicate. The Address is composed of two slots, "Street" and "City"; both are mandatory and the permitted value is a String. The Person, instead, is composed of three slots, "Name" "Age" and "Address", where the type of name is a string, the type of age is an integer, and the type of address (that is an optional slot in the Person frame) is another concept, specifically the "Address" type of concept.

A more interesting alternative is when a factory of application-specific objects is registered with the ontology for each entity in the domain of discourse, as shown in the following example. This alternative allows a programmer to work directly with application-specific classes (like Company, EngageAction, …), while the content language encoding and the Frame class is used only by the JADE framework.

```
class CompanyFactory implements jade.onto.RoleFactory {
  public Object create(Frame f)   { return new Company(); }
  public Class  getClassForRole() { return Company.class; }
}

class Company {
    private String name;
    private Frame  headquarter;
    public void   setName(String n)      {name = n;}
    public String getName()              {return name;}
    public void   setHeadquarter(Frame a){headquarter = a;}
    public Frame  getHeadquarter()       {return headquarter;}
}

myOntology.addFrame("Company", Ontology.CONCEPT_TYPE,
  new TermDescriptor[] {
    new TermDescriptor("name", Ontology.STRING_TYPE, Ontology.M),
    new TermDescriptor("headquarter", Ontology.CONCEPT_TYPE, "Address",
                          Ontology.M)
                                     }, new CompanyFactory()
                                        });
```

In the example above, an entity named "Company" is added to the Ontology. This entity has two mandatory slots: the "name", that is a String, and the "headquarter" that assumes values of type "Address" (where "Address" is a name of entity registered before). The role of this entity is implemented by the *Company* class, where the class *CompanyFactory* is the factory of *Company* objects (i.e. when called it returns a new object of type *Company*) that is registered with and used by the ontology. Notice that, the supplied class Company must obey to some rules, in order to be accepted by the Ontology object:

- for every *TermDescriptor* object of the array, of type *T* and name *XXX*, the class must have two accessible methods, with the following signature:
  - T getXXX()
  - void setXXX(T value)
- furthermore, in order to play the role of an action, the class must also have two accessible methods, with the following signature, to get and set the name of the agent name that has to perform the action:
  - String getActor()
  - void setActor(String actor)

As a useful technique, one can define compliant Java interfaces and add them to the Ontology; this way useful OO idioms such as polymorphism and mix-in inheritance can be exploited for the Java representations of ontological objects. Notice that an ontology can be composed of a mix of application-specific objects (e.g. Company) and entities with no factory associated; however there is no expected advantage into using such a non-uniform style.

Due to different lexical conventions between the Java language, FIPA ACL, and content languages, some name translation is performed to map the name of a term (that is, of a frame slot or of an action argument) into the name of the corresponding method, in particular ':' and '-' characters are removed, and case insensitive match is performed.

In the following, the example continues by adding the action "Engage" to the ontology.

```
class EngageActionFactory implements jade.onto.RoleFactory {
  public Object create(Frame f)   { return new EngageAction(); }
  public Class  getClassForRole() { return EngageAction.class; }
}

class EngageAction {
    private Frame personToEngage;
    private Company engager;
    private String actor;
    public void   set_0(Frame desc)  {personToEngage = desc;}
    public Frame  get_0()            {return personToEngage;}
    public void   set_1(Company desc){engager = desc;}
    public Frame  get_1()            {return engager;}
    public void   setActor(String a) {actor = a;}
    public String getActor()         {return actor;}
}

myOntology.addFrame("engage", Ontology.ACTION_TYPE,
  new TermDescriptor[] {
    new TermDescriptor(Ontology.CONCEPT_TYPE, "Person", Ontology.M),
    new TermDescriptor(Ontology.CONCEPT_TYPE, "Company", Ontology.M)
                                }, new EngageActionFactory()
                                   });
```

This entity is an Action and has two mandatory arguments. The type of the first argument is a Concept of type "Person", while the type of the second is a *Company*. The role of this action is implemented by the *EngageAction* class and the class *EngageActionFactory* is the factory of *EngageAction* objects. The following additional things must be noticed in this example: a) because the class *EngageAction* represents an action, the framework requires the two methods *setActor* and *getActor*; b) in order to allow the content language codec to set and get unnamed arguments (i.e. the semantics of the arguments is based only on their order), the framework requires the methods *set_0, get_0, set_1,* and *get_1,* that is a couple of methods for each additional argument with the position of the argument at the end of the name (e.g. *set_0, set_1, set_2, …*); c) because there is no factory registered with the ontology for the entity "Person", *EngageAction* deals with *Frame* arguments rather than application-specific arguments.

Having added all the entities to the ontology, the ontology object can be registered with the Agent by using the method *registerOntology* available in the *Agent* class.

```
myAgent.registerOntology("sample-1", myOntology);
```

It is important to notice that two agents, who wish to converse, are not required to share the same Java objects (e.g. *Company, EngageAction, …),* but just a mutual understanding of the domain symbols (e.g. the strings "Person", "Company", "Address", "headquarter", …). The usage of the Java objects is just a way to integrate ontologies with the Java language.

3.6.3 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 Java object 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 Java object 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 3.6 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
`:name`	**`String`**	**Mandatory**
`:address`	**`String`**	**Mandatory**

`:destination`	**`:location`**	**Mandatory**
`:agent-profile`	`:mobile-agent-profile`	Optional
`:agent-version`	`String`	Optional
`:signature`	`Binary`	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
`:system`	`:mobile-agent-system`	Optional
`:language`	`:mobile-agent-language`	Optional
`:os`	**`:mobile-agent-os`**	**Mandatory**

- `: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
`:name`	**`String`**	**Mandatory**
`:major-version`	**`Short`**	**Mandatory**
`:minor-version`	`Short`	Optional
`:dependencies`	`String`	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
`:name`	**`String`**	**Mandatory**
`:major-version`	**`Short`**	**Mandatory**
`:minor-version`	`Short`	Optional
`:dependencies`	`String`	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
`:name`	**`String`**	**Mandatory**
`:major-version`	**`Short`**	**Mandatory**
`:minor-version`	`Short`	Optional
`:dependencies`	`String`	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
:name	**String**	**Mandatory**
:transport-protocol	**String**	**Mandatory**
: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 `String`. 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. As soon as the FIPA 99 specifications for the SL-0 language will be release, the grammar will also be updated to comply with it.

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 *SL0* 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 da0
 :receiver ams
 :content
 ( action ams ( move-agent
  ( :mobile-agent-description
    ( :name Peter )
    ( :address IOR:000… )
    ( :destination
       ( :location
          ( :name Front-End )
          ( :transport-protocol JADE-IPMT )
          ( :transport-address IOR:000…Front-End )
        )
    )
  )
  ) )
 :language  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 3.6.

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 `String` argument, holding the name 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  da0
```

```
    :receiver ams
    :content ( action ams ( where-is-agent Peter ) )
    :language SL0
    :ontology jade-mobility-ontology
    :protocol fipa-request
   )
```

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

```
   ( INFORM
    :sender ams
    :receiver da0
    :content
   ( :location
     ( :name Front-End )
     ( :transport-protocol JADE-IPMT )
     ( :transport-address IOR:000….Front-End )
   )
    :language SL0
    :ontology jade-mobility-ontology
    :protocol fipa-request
   )
```

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

```
   ( REQUEST
    :sender Johnny
    :receiver AMS
    :content ( action AMS ( query-platform-locations ) )
    :language SL0
    :ontology jade-mobility-ontology
    :protocol fipa-request
   )
```

Since the SL0 language, as defined by FIPA 97 standard, lacks a way to represent multi-slots, JADE had to use a workaround for this. The `:content` slot of the `inform` message is a single string, holding the list of all the available locations, separated by “||”. If the current platform has three containers, the AMS sends back the following `inform` message:

```
( INFORM
 :sender AMS
 :receiver Johnny
 :content
  ( :location
    ( :name Container-2 )
    ( :transport-protocol JADE-IPMT )
    ( :transport-address IOR:000….Container-2 ) )
||( :location
    ( :name Container-1 )
    ( :transport-protocol JADE-IPMT )
    ( :transport-address IOR:000….Container-1 ) )
||( :location
    ( :name Front-End )
    ( :transport-protocol JADE-IPMT )
    ( :transport-address IOR:000….Front-End ) )
 :language  SL0
 :ontology  jade-mobility-ontology
 :protocol  fipa-request
)
```

JADE provides the `MobilityOntology.parseLocationList()` static method, that takes the *SL0* `Codec` and the message content as arguments and returns an array of `MobilityOntology.Location` objects.

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.

5. RUNNING THE AGENT PLATFORM

To use the framework at least Java Development Kit version 1.2 is required. To build the system, one needs also 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. However, pre-built IDL stubs and Java parser classes are included with the JADE source distribution.

5.1 Getting the software

First of all, you can download JADE in source form and recompile it yourself, or get the pre-compiled binaries (actually they are JAR files). Moreover, some sample agents and a demo application (a meeting scheduler) are provided in a separate package. The documentation for JADE (a set of HTML pages and a Programmer's Guide) can be downloaded separately, too.

5.2 Running JADE from the binary distribution

After uncompressing the archive, you will have a directory tree starting with `jade` with a `lib` subdirectory, containing some JAR files. You just have to add all the JAR files to your `CLASSPATH`, and you are ready to start JADE.

To run the Agent Platform, one must issue the command:

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

Use -h option to get a list of command line arguments.

To start some agents on additional hosts, one must create and run more Agent Containers; these components connect themselves with the main 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 [options] [Agent list]
```

Again, using -h command-line option explains program usage.

As can be seen from above, the same command is issued in the two cases, with the `-platform` command-line switch used to choose between an ordinary Agent Container and the global Agent Platform. Using command-line options, users can state host name and port number where the main Agent Platform resides, and the name with which it is registered in RMI Registry. This way, multiple platforms can be executed on a single host.

The agent list is a sequence of character strings; each of them must be broken in two parts by a colon ':'. 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.

Refer to the `README` file in `src/examples` directory to get some explanations of each example program behaviour.

5.1.1 Example

First of all set the `CLASSPATH` to include the JAR files in the `lib` subdirectory and the current directory (for Windows 9x/NT: `set CLASSPATH=%CLASSPATH%;.;c:\jade\lib\jade.jar;c:\jade\lib\jadeTools.jar;c:\jade\lib\Base64.jar`) then execute the following commands.

Start the platform:

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

and start some agent containers on other shells. For example start an agent container 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
```

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

If you like, you can 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
```

sender2 " (`examples.receivers.AgentSender` is the code that implements it) and receiver2 (`examples.receivers.AgentReceiver` is the code that implements it) are the names of the two agents.

5.3 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 you can 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.

You can perform the following build operations:

Building JADE platform

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.

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 5.2 for running 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 wuth empty JAR files.

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).

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 jessexample
```

with makefiles, or

```
makejessexample
```

with batch files.

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.

5.4 IIOP support and inter-platform messaging

To be fully FIPA compliant, JADE has IIOP capabilities. Using IIOP protocol, it is possible to connect multiple agent platforms, either many instances of JADE running on different host/port or a JADE platform 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. FIPA 97 specifications suggest a URL notation to name an agent in a world-wide unique way: for example an agent can be named `peter@iiop://fipa.org:50/acc`. When using IIOP the URL must be mapped into a regular CORBA object reference: while mapping for URL host and URL port are obvious, FIPA 97 states that URL file name ('acc') must be mapped into the *object key* for the CORBA Object Implementation of `FIPA_Agent_97` interface representing the agent platform. Unfortunately, most of the CORBA ORB implementations available do not allow users to choose meaningful words such as 'acc' as object keys; rather, they create some hash values arbitrarily and this results in URL strings with binary characters within their file part, which are considered incorrect by FIPA 97 grammar for agent addresses.

To overcome this limitation, 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:000000000000001649444c644f4`…

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).

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 GUID is made by a local name, the '@' character, and the platform IOR; the platform IOR can be used (through cut and paste) to compose an ACL message (typically a DF registration message), thus connecting two platforms. Since JADE attaches the platform IOR to all outgoing messages, there's no need to put it in `:sender` field; just the local agent name will be enough. Besides, calling `getAddress()` on any agent returns the IOR of the platform that agent belongs to.

For example, an agent could read the IOR of a remote platform and then register itself with the remote DF with the following code:

```
String remoteIOR = … // Read from File, from a Socket, input from user
registerWithDF("df@" + remoteIOR);
```

Using the DF administrative GUI, one can easily federate DF agents from different JADE platforms by pasting an IOR into the text field that asks for DF name and prepending it with "df@" (or whatever the remote DF agent is called).

6. RELEASE NOTES

6.1 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.

6.2 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.

6.2 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.

6.3 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.

6.4 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 how 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.

6.5 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.

6.6 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.

6.7 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.

6.8 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.

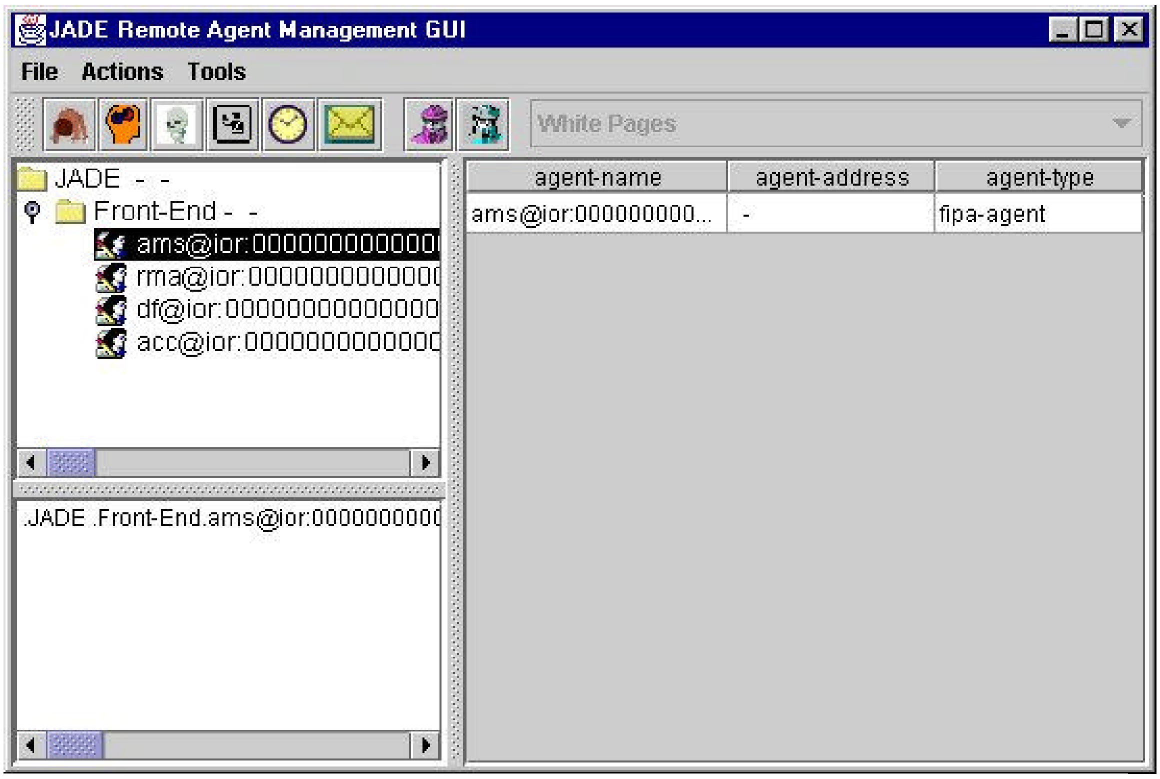
7. GRAPHICAL USER INTERFACE TO MANAGE AND MONITOR THE AP ACTIVITY

JADE offers a graphical interface to platform administration through its RMA agent; this agent shows the state of the Agent Platform it belongs to (agents and agent containers) and offers various tools to request administrative action from the AMS agent and to debug and test JADE-based applications.

An RMA is a Java object, instance of the class `jade.tools.rma.rma` and can either be started on the command-line as an ordinary agent (i.e. with the command `java jade.Boot myConsole:jade.tools.rma.rma`), but an instance named just '`rma`' can also be launched by supplying the `'-gui`' argument on the command line parameters (i.e. `with java jade.Boot –gui` command). More than one RMA can be started on the same platform as long as every instance has a different local name, but only one for a given agent container.

Note: this limitation is currently due to implementation reasons that will soon be removed. For now, running more than one RMA within the same JVM will cause JADE to malfunction; see also BUGS section in the README file.

The figure below shows the RMA GUI, when a JADE platform composed of two containers is started.



agent-name	agent-address	agent-type
ams@ior:000000000...	-	fipa-agent

Figure 1 – The RMA graphical console

The RMA GUI offers access to platform features using the menu bar and the tool bar visible in the figure, but also provides context-sensitive popup menus and tool tips (one of which can be seen in the picture, indicating that the DF is running). Now a list of the menu options available will be presented, along with the toolbar buttons that can be used to perform it, if any.

- `File menu:`

  This menu contains the general commands to the RMA.

  - `Close RMA Agent`

    This item terminates the RMA agent, invoking its `doDelete()` method, but leaves all the rest of the platform unaffected. Closing the RMA window has the same effect as invoking this command.

  - `Exit this Container`

    This item terminates the agent container the RMA is living in, killing the RMA and all the other agents in the process. If this container is the Agent Platform Front-End, then the whole platform is shut down.

  - `Shut down Agent Platform`

    This item closes down the whole agent platform, terminating all connected containers, then all user agents present on platform front end, and eventually JADE system 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 using the current selection of the agent tree as the target; most of these actions also are associated to 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 or she wants the agent to start on. If no container is specified, the agent is launched on the Agent Platform Front-End. The leftmost button of the toolbar ( )performs this action.

♦ `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 agent it holds and then deregisters that container from the platform. Of course, if the Agent Platform Front-End is currently selected, then the whole platform is shut down. The third button of the toolbar ( ) fires this action.

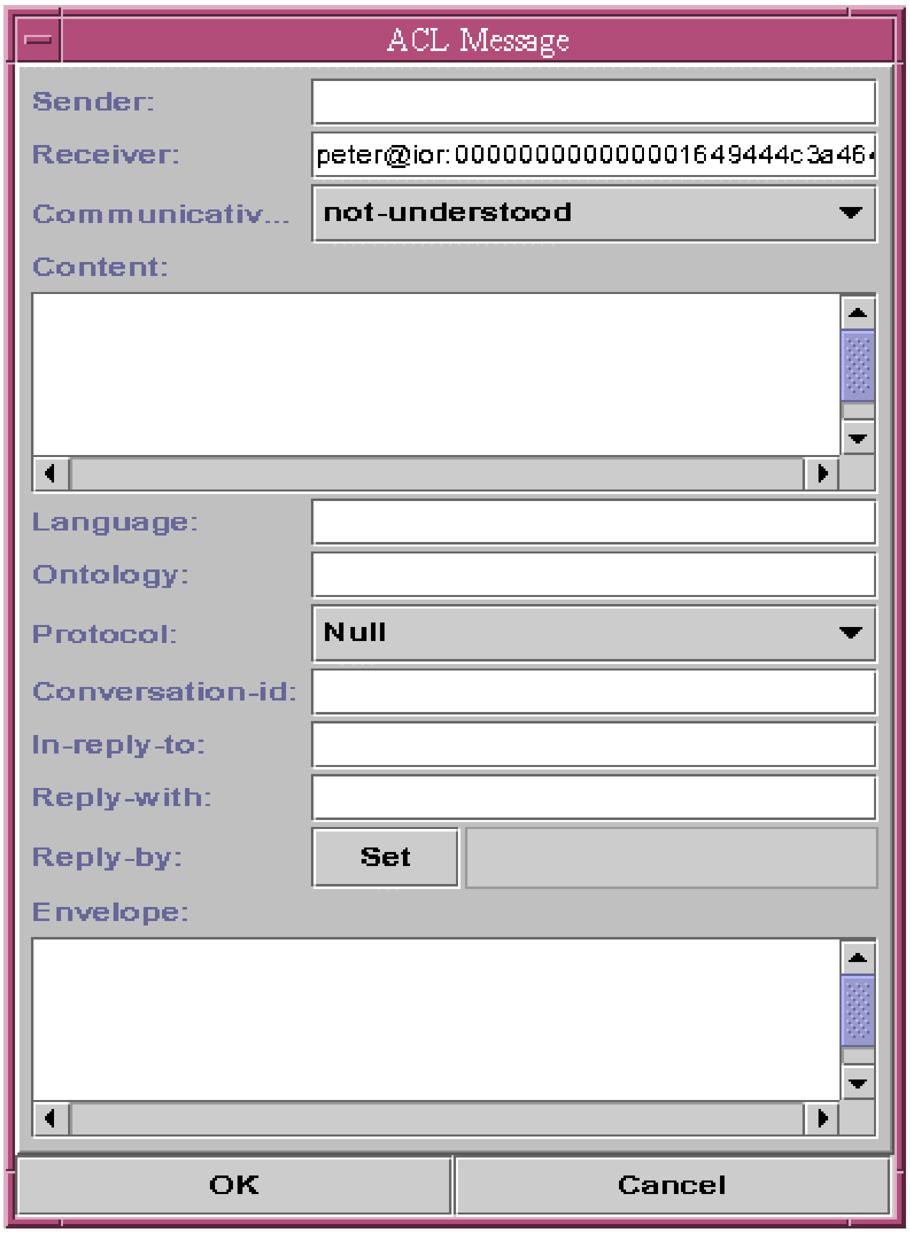
♦ `Suspend Selected Agents`

This action suspends the selected agents and is equivalent to calling its `doSuspend()` method. Beware that suspending a system agent, particularly the AMS, deadlocks the platform, since the RMA acts as a client and is the AMS who ultimately carries out all requested administrative operations. This action can be requested clicking on the fourth ( ) button of the toolbar.

♦ `Resume Selected Agents`

This action puts the selected agents back into th `AP_ACTIVE` state, provided they were actually suspended, and works just the same as calling their `doActivate()` method. It is linked to the fifth ( ) button in the toolbar.

♦ `Send Custom Message to Selected Agents`



Sender:	
Receiver:	peter@ior:0000000000000001649444c3a46
Communicativ...	not-understood
Content:	
Language:	
Ontology:	
Protocol:	Null
Conversation-id:	
In-reply-to:	
Reply-with:	
Reply-by:	
Envelope:	

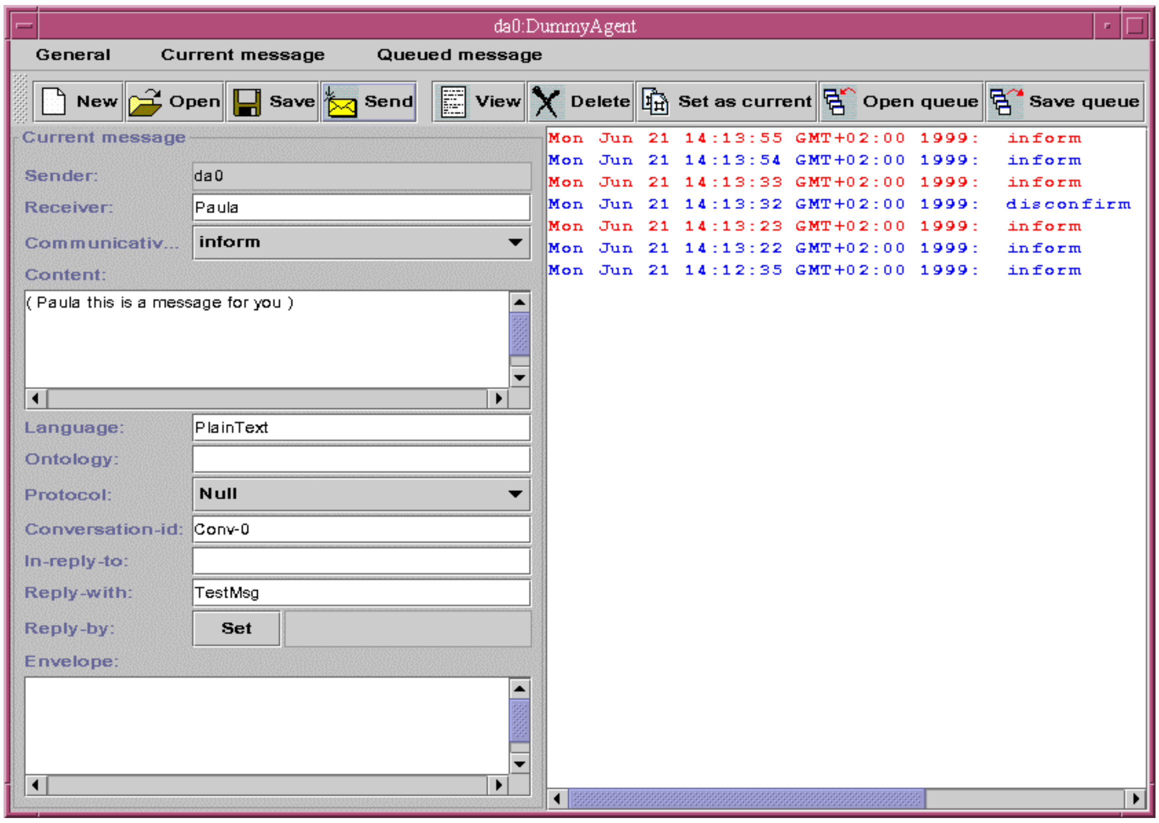
This action allows to send an ACL message to an agent. Currently, multiple receivers are not supported. When the user selects this menu item (or clicks on the next to rightmost button of the toolbar ✉), a special dialog is displayed in which an ACL message can be composed and sent, as the figure below shows.

- `Tools menu:`

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

♦ `Start DummyAgent`

The DummyAgent tool allows users to interact with JADE agents hiding themselves behind a special agent.DummyAgent's GUI allows to compose and send ACL messages, but also keeps 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 DummyAgent can be started from this menu item or from the associated toolbar button ( ). The following picture shows the DummyAgent in action.



Sender:	da0
Receiver:	Paula
Communicativ...	inform
Content:	(Paula this is a message for you)
Language:	PlainText
Ontology:	
Protocol:	Null
Conversation-id:	Conv-0
In-reply-to:	
Reply-with:	TestMsg
Reply-by:	Set
Envelope:	

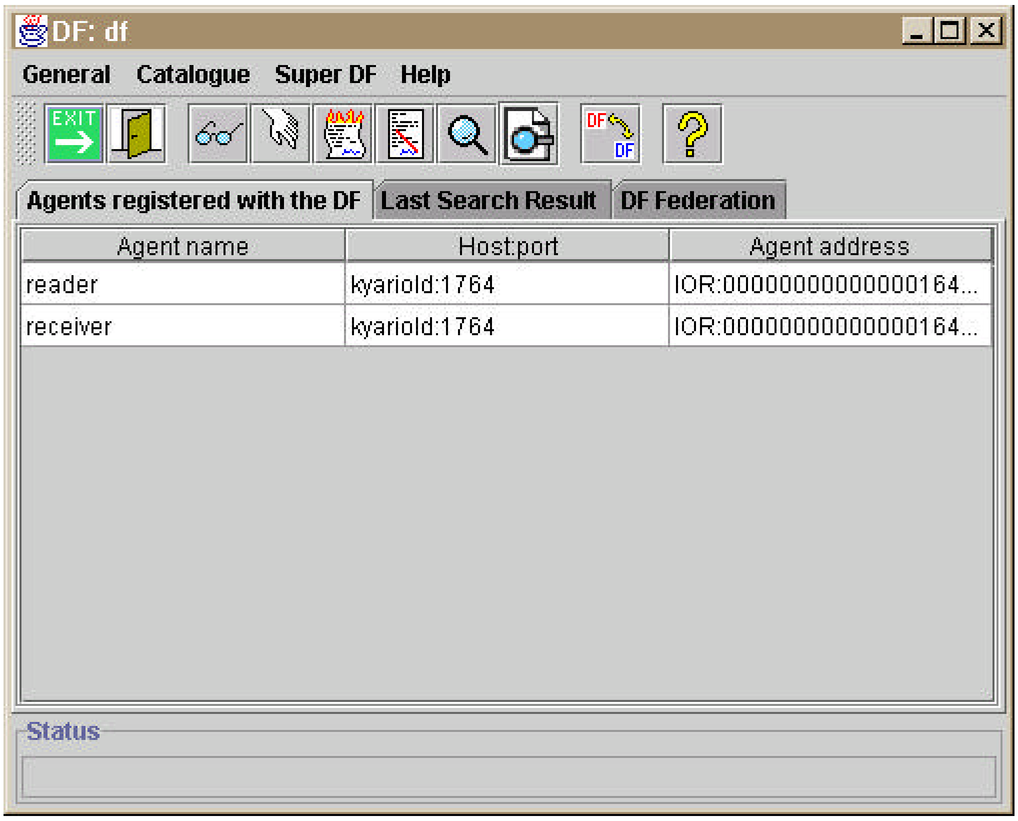
```
Mon  Jun  21  14:13:55  GMT+02:00  1999:   inform
Mon  Jun  21  14:13:54  GMT+02:00  1999:   inform
Mon  Jun  21  14:13:33  GMT+02:00  1999:   inform
Mon  Jun  21  14:13:32  GMT+02:00  1999:   disconfirm
Mon  Jun  21  14:13:23  GMT+02:00  1999:   inform
Mon  Jun  21  14:13:22  GMT+02:00  1999:   inform
Mon  Jun  21  14:12:35  GMT+02:00  1999:   inform
```

♦ `Show the DF GUI`

This item of the menu allows to activate the GUI of the default DF of the platform. Notice that this GUI is actually executed on the host where the platform (front-end container) was executed.

Using this GUI the user can interact with the DF in order to view the descriptions of the registered agents, to register/deregister agents with this DF, to modify the description of an agent, to search for agent descriptions, and to make a federation of DFs.

This GUI shows three different views of the functions provided by a DF. According to the view selected only some actions are allowed. An on-line help provides a detailed guide to the use of this GUI.



Agent name	Host:port	Agent address
reader	kyariold:1764	IOR:0000000000000164...
receiver	kyariold:1764	IOR:0000000000000164...

Figure 3 - GUI of the DF

The figure shows the DF GUI.

Version	date	author	Changes
0.1	19/7/98	Paolo Marenzoni	
0.2	21/7/98	Fabio Bellifemine	Added FAPF features, rewritten "How to implement an agent"
0.3	28/7/98	Fabio Bellifemine	Name is now JADE. Included first 3 sections of JADE Package. Rewritten "A sample agent"
0.4	3/8/98	Paolo Marenzoni	Added section "The Agent class" (still incomplete)
0.5	14/09/98	Paola Turci	Added sections "Implementing the Agent behaviour", "A sample Agent System" (still incomplete) and "Running the Agent Platform"; added section "Using the Agent class" resulted from reorganising sections "A sample Agent" and "A sample Agent behaviour"
0.6	16/9/98	Fabio Bellifemine	Modified the sections "Using the Agent class", "Running the Agent Platform", "Implementing the Agent behaviour", "The ACLMessage class".
0.7	20/9/98	Paola Turci	Completed the section "Using the Agent class"
0.8	23/9/98	Paola Turci	Completed the section "Implementing the Agent behaviour "
0.9	23/9/98	Fabio Bellifemine	Minor modifications. Inserted comments where clarifications is needed.
1.0	29/9/98	Fabio Bellifemine	Removed comments. Documented block(). Some minor modifications. Sent with Jade 0.79+.
1.1	5/10/98	Fabio Bellifemine	Changed the introduction with new text from Giovanni Rimassa. Included UML picture of the Behaviour hierarchy.
1.2	5/11/98	Fabio Bellifemine	Changed the name in Java Agent Development Framework instead of kit. Added release notes of 0.9
1.3	21/12/98	Giovanni Rimassa	General review to update Programmer's Guide. Added sections to explain FIPA system agents usage.
1.4	15/2/99	Giovanni Rimassa	Updated documentation to JADE 0.97 . Added section on IIOP usage.
1.5	17/2/99	Fabio Bellifemine	Restyling. Minor corrections. Added interaction protocols.
1.6	21/6/99	Giovanni Rimassa	Update for version 1.0 of JADE.
1.7	22/6/99	Fabio Bellifemine	Added some interaction protocols.
1.8	1/10/99	Fabio Bellifemine	Documented the new Base64 encoding feature in the ACLMessage class.
1.9	3/11/99	Fabio Bellifemine	Documented the new features of the ACLMessage class (where the performative is now an int) and improved the documentation of the interaction protocols.
1.10	22/11/99	Fabio Bellifemine	Major changes in JADE 1.2
1.11	24/2/2000	Giovanni Rimassa	General review for JADE 1.3. Removed class reference part (now in HTML format).
1.12	24/2/2000	Fabio Bellifemine	Added release notes for JADE 1.3.
1.13	15/3/2000	Fabio Bellifemine	corrected example according to problem reported by Sven Mecke
	2/6/2000	Fabio Bellifemine	Added description of the content language and ontology support. Added release notes of JADE 1.4 Documented makefile for JESS.
	5/6/2000	Giovanni Rimassa	Added description of the support for intra-platform mobility.