JADE ADMINISTRATOR'S GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

Last update: 28-Ferbruary-2006. JADE 3.4

Authors: Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT)
        Giovanni Rimassa (FRAMeTech s.r.l.), Roland Mungenast (PROFACTOR GmbH)

TABLE OF CONTENTS

1   INTRODUCTION

This administrator's guide describes how to install and launch JADE. It is complemented by the HTML documentation available in the directory jade/doc and the JADE Programmer's Guide. 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.

2   RUNNING THE AGENT PLATFORM

2.1   Software requirements

The only software requirement to execute the system is the Java Run Time Environment version 1.4.

In order to build the system the JDK1.4 is sufficient because pre-built IDL stubs and Java parser classes are included with the JADE source distribution. Those users, who wish to regenerate IDL stubs and Java parser classes, should also have the JavaCC parser generator (version 0.8pre or version 1.1; available from http://www.metamata.com), and the IDL to Java compiler (available from the Sun Developer Connection). Notice that the old idltojava compiler available with JDK1.2 generates wrong code and it should never be used, instead the new idlj compiler, that is distributed with JDK1.3, should be used.

2.2   Getting the software

All the software is distributed under the LGPL license limitations and it can be downloaded from the JADE web site http://jade.tilab.com/. Five compressed files are available:

1. The source code of JADE
2. The source code of the examples
3. The documentation, including the javadoc of the JADE API and this programmer's guide
4. The binary of JADE, i.e. the jar files with all the Java classes
5. A full distribution with all the previous files

2.3   Running JADE from the binary distribution

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

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

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

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

An Agent Container can be started using the command:

```
java jade.Boot –container [options] [AgentSpecifier list]
```

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

```
java –jar lib\jade.jar –nomtp [options] [AgentSpecifier list]
```

Remind to use the "–nomtp" option, otherwise an exception will be thrown because the library http.jar is not found.

2.3.1 Command line syntax

The full EBNF syntax of the command line is the following, where common rules apply for the token definitions:

```
java jade.Boot Option* AgentSpecifier*


Option           = "-container"
                 | "-backupmain"
                 | "-gui"
                 | "-nomtp"
                 | "-services" ClassName (";" ClassName )*
                 | "-host" HostName
                 | "-port" PortNumber
                 | "-local-host" HostName
                 | "-local-port" PortNumber
                 | "-name" PlatformName
                 | "-container-name" ContainerName
                 | "-smaddrs" HostName (":" PortNumber )? (";" HostName
(":" PortNumber )? )*
                 | "-mtp" ClassName "(" Argument* ")"
                    (";" ClassName "(" Argument* ")")*
                 | "-aclcodec" ClassName (";" ClassName )*
                 | "-nomobility"
                 | "-version"
                 | "-help"
                 | "-conf" FileName?
                 | "-" Keyword Value


ClassName        = PackageName? Word

PackageName      = (Word ".")+

Argument         = Word | Number | String

HostName         = Word ( "." Word )*

PortNumber       = Number

AgentSpecifier = AgentName ":" ClassName ("(" Argument* ")")?

AgentName        = Word

PlatformName     = Word
```

```
Keyword         = Word

Value           = Word
```

2.3.2 Options available from the command line

-container specifies that this instance of JADE is a container and, as such, that it must join with a main-container (by default this option is unselected)

-host specifies the host name where the main container to register with is running; its value is defaulted to localhost. This option can also be used when launching the main-container in order to override the value of localhost; **a typical example of this kind of usage is to include the full domain of the host (e.g. *–host kim.cselt.it* when the localhost would have returned just *'kim'*) such that the main-container can be contacted even from outside the local domain.**

-port this option allows to specify the port number where the main container to register with is running. By default the port number 1099 is used.

-gui specifies that the RMA (Remote Monitoring Agent) GUI of JADE should be launched (by default this option is unselected)

-local-host specifies the host name where this container is going to run; its value is defaulted to localhost. **A typical example of this kind of usage is to include the full domain of the host (e.g. *–host kim.cselt.it* when the localhost would have returned just *'kim'*) such that the main-container can be contacted even from outside the local domain.**

-local-port this option allows to specify the port number where this container can be contacted. By default the port number 1099 is used.

-name this option specifies the symbolic name to be used as the platform name; this option will be considered only in the case of a main container; the default is to generate a unique name from the values of the main container's host name and port number. Please note that this option is strongly discouraged since uniqueness of the HAP is not enforced. This might result in non-unique agent names.

-container-name this option specifies the symbolic name to be used as the name of this container.

-services specifies a list of classes, implementing JADE kernel-level services, that must be loaded and activated during startup. Class names are to be separated by semicolon, and must be fully qualified. The special messaging and management services are always activated and need not be specified with this option. If this option is not given, by default the mobility and event notification services are started. The following table lists the services available with the standard distribution of JADE. More services (e.g. supporting security and agent persistence are available as add-ons)

Name	Service Class	Features
Messaging	jade.core.messaging.MessagingService	ACL message exchange and MTP management. **Activated automatically**
Agent-Management	jade.core.management.AgentManagementService	Basic management of agent life-cycle. Container and platform shutdown. RMA support. **Activated automatically**
Agent-Mobility	jade.core.mobility.AgentMobilityService	Agent mobility support. **Active by default**
Notification	jade.core.event.NotificationService	Support for platform-level event system notifications. This is required to support the Sniffer and Introspector tools (see 6.4and 6.5). **Active by default**
Persistent-Delivery	jade.core.messaging.PersistentDeliveryService	Allows buffering and persistent storage for undelivered ACL messages (see 5 for details). **Inactive by default**
Main-Replication	jade.core.replication.MainReplicationService	Support for replicating the Main Container for fault tolerance purposes. Has to be activated on each node hosting a Main Container (see 4 for details). **Inactive by default**
Address-Notification	jade.core.replication.AddressNotificationService	Support for being notified whenever the list of active Main Container copies changes, i.e. a new copy is added or an existing copy terminates (see 4 for details). **Inactive by default**
UDPNodeMonitoring	Jade.core.nodeMonitoring.UDPNodeMonitoringService	Support for monitoring of platform nodes by means of UDP packets (see 2.3.7 for

		details)
		Inactive by default

-*mtp*   specifies a list of external Message Transport Protocols to be activated on this container (by default the JDK1.4 IIOP is activated on the main-container and no MTP is activated on the other containers)

-*nomtp*   has precedence over -mtp and overrides it. It should be used to override the default behaviour of the main-container (by default the -nomtp option unselected)

-*backupmain* specifies that this instance of JADE is a backup main container (see 4) and, as such, that it must join with a main-container (by default this option is unselected)

-*smhost*   when a Main Container copy is started, this option must be used to select the host name where the Service Manager is to be exported (typically it is the local host). This option is only useful on nodes where the Main-Replication service has been activated.

-*smport*   when a Main Container copy is started, this option must be used to select the TCP port where the Service Manager is to be exported (typically it is the local host). This option is only useful on nodes where the Main-Replication service has been activated.

-*smaddrs*   when multiple Main Container copies are active in the platform, a peripheral container connects to the one pointed to by the <-host, -port> option pair. With this option, the addresses of all the other copies can be listed, so that the container is able to re-connect to another copy should the current one become unavailable. This option is an alternative to activating the Address-Notification service: instead of setting up an update protocol, one simply provides a fixed list of well-known addresses for Main Container copies.

-*aclcodec* By default all messages are encoded by the String-based ACLCodec. This option allows to specify a list of additional ACLCodec that will become available to the agents of the launched container in order to encode/decode messages. JADE will provide automatically to use these codec when agents set the right value in the field *aclRepresentation* of the Envelope of the sent/received ACLMessages. Look at the FIPA specifications for the standard names of these codecs

-*nomobility* disable the mobility and cloning support in the launched container. In this way the container will not accept requests for agent migration or agent cloning, option that might be useful to enhance the level of security for the host where this container is running. Notice that the platform can include both containers where mobility is enabled and containers where it is disabled. In this case an agent that tries to move from/to the containers where mobility is disabled will die because of a Runtime Exception.

Notice that, even if this option was selected, the container would still be able to launch new agents (e.g. via the RMA GUI) if their class can be reached via the local CLASSPATH.
By default this option is unselected.

-version   print on standard output the versioning information of JADE (by default this option is unselected)

-help   print on standard output this help information (by default this option is unselected)

-conf   if no filename is specified after this option, then a graphical interface is displayed that allows to load/save all the JADE configuration parameters from a file. If a filename is specified, instead, then all the options specified in that file are used to launch JADE. By default this option is not selected

Other properties can also be set via the command line via the syntax "–keyword value" (see also the tutorial on How to pass arguments and properties to agents).

2.3.3 Launching agents from the command line

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

Each string is broken into three parts. The first substring (delimited by the colon ':' character) is taken as the agent name; the remaining substring after the colon (ended with a space or with an open parenthesis) is the name of the Java class implementing the agent. The Agent Container will dynamically load this class. Finally, a list of string arguments can be passed delimited between parentheses.

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.

Another example is the string `Peter:myAgent("today is raining" 123)` that means "create a new agent named Peter whose implementation is an object of class `myAgent` and pass an array of two arguments to its constructor: the first is the string `today is raining` and the second is the string 123". Notice that, according to the Java convention, the quote symbols have been removed and the number is still a string.

2.3.4 Example

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

```
set CLASSPATH=%CLASSPATH%;.;c:\jade\lib\jade.jar;
c:\jade\lib\jadeTools.jar;c:\jade\lib\Commons-codec\commons-
codec-1.3.jar;c:\jade\lib\iiop.jar
```

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

```
prompt> java jade.Boot –gui
```

Execute the following command to start an agent container on another machine, by telling it to join the Agent Platform 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 -host kim.cselt.it -container
           sender1:examples.receivers.AgentSender
```

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

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

```
prompt> java jade.Boot –host kim.cselt.it –container
           receiver2:examples.receivers.AgentReceiver
           sender2:examples.receivers.AgentSender
```

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

2.3.5 Starting Jar-packaged agents

Until version 3.3 of JADE, class files belonging to user-created agents and the platform itself were mixed in the classpath. There was no way to distinguish between agent's code and platform's code. Furthermore, agent and platform classes are loaded using the same ClassLoader. This implies that all agents and the platform share the same class namespace and two agents cannot use the same agent class name.

In order to start this differentiation the concept of "JAR agent" has been introduced since version 3.4 of the JADE platform. Jar Agents are agents whose code, as the name indicates, is packaged inside a JAR file. Those jar files must not be in the classpath, they must be placed in a special folder which can be specified as follows:

```
java jade.Boot -jade_core_management_AgentManagementService_agentspath
c:\tmp\jarsfolder
```

Once the path is specified, we can put jar files containing agent code inside the folder, that is, one jar per agent. In order to start correctly this kind of agents, at the moment, we adopt the convention of naming jar files like agent's main class. If we use packages we must substitute the dots by underscore character. As an example, suppose that we have an agent which main class is coffeeApp.agents.WaiterAgent. If we want to load this agent as a Jar Agent, we must pack all the agent code in a single jar file named coffeeApp_agents_WaiterAgent.jar and place it in the previously specified folder. The process of starting this agent is the usual one via gui or command line. The main difference now is that the agent must not be in the classpath and will be loaded in a separate namespace.

If the parameter jade_core_management_AgentManagementService_agentspath is not specified,

the working directory will be used by this purpose.

2.3.6 Configuring the Yellow Pages service provided by the DF agent

The DF agent, that provides the default Yellow Pages service in accordance to the FIPA specifications, accepts a number of optional configuration parameters that can be set either as command line options or within a properties file (to be passed to the DF as an argument).

`jade_domain_df_autocleanup`	When set to `true`, indicates that the DF will automatically clean up registrations as soon as agents terminate. (defaults to `false`)
`jade_domain_df_maxleasetime`	Indicates the maximum lease time (in millisecond) that the DF will grant for agent description registrations (defaults to infinite).
`jade_domain_df_maxresult`	Indicates the maximum number of items found in a search operation that the DF will return to the requester. This will only be applied if there are no explicit search constraint specified (defaults to 100).
`jade_domain_df_kb-factory`	This parameter allows specifying the name of the factory class which will be used by the DF to create a knowledge base object for storing its catalogue. (Defaults to: `jade.domain.DFKBFactory`) The specified class has to be a sub class of `jade.domain.DFKBFactory`.

By default the DF stores its catalogue in memory. Through the following optional parameters the DF can also be configured to store its catalogue directly into a database:

`jade_domain_df_db-default`	If set to `true`, indicates that the DF will store its catalogue into an HSQLDB database, which is started in the same JVM as JADE. This is the easiest way to persist the DF catalogue. Apart from using this parameter only the HSQLDB class files have to be added to the Java CLASSPATH. Any further configuration is done automatically by JADE. HSQLDB is not part of the JADE distribution but can be downloaded from the HSQL Database Engine project at the web site: http://sourceforge.net/projects/hsqldb.
`jade_domain_df_db-url`	Defines the JDBC URL of the database the DF will store its catalogue into. With this parameter the DF can be configured to use other databases than HSQLDB.
`jade_domain_df_db-cleantables`	If set to `true`, indicates that the DF will clean the content of all pre-existing database tables, used by the DF. This parameter is ignored if the catalogue is not stored in a database. (defaults to `false`)

11

Together with the `jade_domain_df_db-url` the following optional parameters can be used to configure the database access:

`jade_domain_df_db-driver`	Indicates the JDBC driver to be used to access the DF database (defaults to the ODBC-JDBC bridge).
`jade_domain_df_db-username`	Indicate the username and password to be used to access the DF database (defaults to null).
`jade_domain_df_db-password`	Indicate the username and password to be used to access the DF database (defaults to null).

Examples

DF with default database

The following command line will launch a JADE main container with a DF that will store its catalogue into a HSQLDB database. The database will be started automatically together with JADE, provided that the HSQLDB libraries can be found in the CLASSPATH.

```
java jade.Boot -jade_domain_df_db-default
```

Using a database over JDBC-ODBC-Bridge

The following command line will launch a JADE main container with a DF that will store its catalogue into a database accessible at URL *jdbc:odbc:dfdb* and that will keep agent registrations for 1 hour at most.

```
java jade.Boot -jade_domain_df_db-url jdbc:odbc:dfdb
-jade_domain_df_maxleasetime 3600000
```

Using a configuration file

The following command line will launch a JADE container with a DF agent called *df1* that will read its configuration from the properties file *df1conf.properties* (located somewhere in the classpath)

```
java jade.Boot –container –host <main-container-host>
df1:jade.domain.df(df1conf.properties)
```

The content of the *df1conf.properties* file may look like that shown in Figure 1.

12

```
#DF configuration file

# DF Catalogue stored in a database (URL is jdbc:odbc:dfdb)
jade_domain_df_db-url = jdbc:odbc:dfdb

# Maximum granted lease time for agent registrations: 1 hour
jade_domain_df_maxleasetime = 3600000
```

Figure 1. Sample DF configuration file

2.3.7 Configuring the Node Failure Monitoring system

Within the JADE platform a main container is constantly monitoring the availability of all its peripheral containers. If a container suddenly crashes the main container recognizes this and automatically removes the container from the platform.

The default JADE failure monitoring system is based on permanent TCP connections towards the monitored remote nodes. If the monitored node crashes the connection is dropped and the main can immediately remove the crashed container from the platform.
The default mechanism is suitable for platforms with a small number of containers and an environment where the interruption of open connections is unlikely. For platforms with a higher number of containers it is recommended to use an alternative UDP based failure monitoring mechanism. This mechanism scales better and in addition to that it allows a container to be unreachable for a certain time.

The UDP based monitoring mechanism can be activated by launching on **both** the main container and peripheral containers the `jade.core.nodeMonitoring.UDPNodeMonitoringService` service using the `-services` option (see 2.3.2).

The following optional parameters can be used on the Main Container (they are automatically propagated to the peripheral containers) to configure it:

`jade_core_nodeMonitoring_UDPNodeMonitoringService_port`	Specifies the port number where the main container will listen for UDP pings. (Default: 28000).
`jade_core_nodeMonitoring_UDPNodeMonitoringService_pingdelay`	Defines the time interval (in milliseconds) between two UDP pings sent by a peripheral container to the main container. (Default: 1.000).
`jade_core_nodeMonitoring_UDPNodeMonitoringService_pingdelaylimit`	Defines the maximum time (in milliseconds) the main container will wait for incoming ping messages before considering a peripheral container "Unreachable". (Default: 3000).
`jade_core_nodeMonitoring_UDPNodeMonitoringService_unreachablelimit`	Defines the maximum time (in milliseconds) a peripheral container can be temporarily unreachable until it gets removed from the platform. (Default: 10.000).

Examples

<u>Main container with UDP monitoring</u>

The following command line will launch a JADE main container with UDP based monitoring, waiting at port 5555 for incoming UDP ping messages. Peripheral containers will send UDP pings every 2 seconds. If the main container doesn't receive a ping message from a peripheral container for more than five seconds this container is marked as temporarily unreachable. If the main container also doesn't receive any ping in the next minute, the peripheral container gets automatically removed from the platform.

```
java jade.Boot
-services jade.core.nodeMonitoring.UDPNodeMonitoringService
-jade_core_nodeMonitoring_UDPNodeMonitoringService_port 5555
-jade_core_nodeMonitoring_UDPNodeMonitoringService_pingdelay 2000
-jade_core_nodeMonitoring_UDPNodeMonitoringService_pingdelaylimit 5000
-jade_core_nodeMonitoring_UDPNodeMonitoringService_unreachablelimit
60000
```

<u>Peripheral Container with UDP monitoring</u>

The following command line will launch a JADE peripheral container with UDP based monitoring (note that all configuration options are specified on the Main Container).

```
java jade.Boot -container
-services jade.core.nodeMonitoring.UDPNodeMonitoringService
```

2.4 Building JADE from the source distribution

If you downloaded the source code of JADE, you can compile it by using the program 'ant', a platform-independent version of make. 'ant' uses the file 'build.xml', which contains all the information about the files that have to be compiled, and that is located into the JADE root directory . The 'ant' program version 1.6 must be installed on your computer, it can be downloaded from the Jakarta Project at the Apache web site: http://jakarta.apache.org/.

2.4.1 Building the JADE framework

In the root directory, just type:

```
ant jade
```

Beware in order to parser classes it is necessary to have JavaCC installed and to set JAVACC_HOME, otherwise the program will produce a comment out.

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.

2.4.2 Building JADE libraries

Just type:

```
ant lib
```

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 2.3 for a description on how to run JADE when you have built the JAR files.

2.4.3 Building JADE HTML documentation

Just type:

```
ant doc
```

You will end up with Javadoc generated HTML pages, integrated within the overall documentation. Note that the generated documentation is not the same that is included in the jadeDoc.zip file that is part of the standard JADE distribution. The latter in fact is a simplified version that cuts out a number of classes and methods that are normally not used and that decrease javadoc readability. This simplified documentation requires the LEAP add-on to be generated. 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).

2.4.4 Building JADE examples and demo application

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

```
ant examples
```

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

```
ant jessexample
```

2.4.5 Cleaning up the source tree

If you type:

```
ant clean
```

you will remove all generated files (classes, HTML pages, JAR files, etc.) from the source tree..

2.5 Support for inter-platform messaging with plug-in Message Transport Protocols

The FIPA 2000 specification proposes a number of different *Message Transport Protocols* (*MTP*s for short) over which ACL messages can be delivered in a compliant way.

JADE comprises a framework to write and deploy multiple *MTP*s in a flexible way. An implementation of a FIPA compliant MTP can be compiled separately and put in a JAR file of its own; the code will be dynamically loaded when an endpoint of that MTP is activated. Moreover, every JADE container can have any number of active MTPs, so that the platform administrator can choose whatever topology he or she wishes.

JADE performs message routing for both incoming and outgoing messages, using a single-hop routing table that requires direct visibility among containers.

When a new MTP is activated on a container, the JADE platform gains a new address that is added to the list in the platform profile (that can be obtained from the AMS using the action

`get-description`). Moreover, the new address is added to all the `ams-agent-description` objects contained within the AMS knowledge base.

2.5.1 Command line options for MTP management

When a JADE container is started, it is possible to activate one ore more communication endpoints on it, using suitable command line options. The `-mtp` option activates a new communication endpoint on a container, and must be given the name of the class that provides the MTP functionality. If the MTP supports activation on specific addresses, then the address URL can be given right after the class name, enclosed in brackets. If multiple MTPs are to be activated, they can be listed together using commas as separators.

For example, the following option activates an IIOP endpoint on a default address.

`-mtp jade.mtp.iiop.MessageTransportProtocol`

The following option activates an IIOP endpoint that uses an ORBacus-based[1] IIOP MTP on a fixed, given address.

```
-mtp
orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cselt.it:12
34/jade)
```

The following option activates two endpoints that correspond to two ORBacus-based IIOP MTP on two different addresses:

```
-mtp
orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cselt.it:12
34/jade);orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cs
elt.it:5678/jade)
```

When a container starts, it prints on the standard output all the active MTP addresses, separated by a carriage return. Moreover, it writes the same addresses in a file, named:

`MTPs-<Container Name>.txt.`

If no MTP related option is given, by default a basic IIOP MTP is activated on the Main Container and no MTP are activated on an ordinary container. To inhibit the creation of the default IIOP endpoint, use the `-nomtp` option.

2.5.2 Configuring MTPs from the graphical management console.

Using the `-mtp` command line option, a transport endpoint lives as long as its container is up; when a container is shut down, all its MTPs are deactivated and the AMS information is updated accordingly. The JADE RMA console enables a more flexible management of the MTPs, allowing activating and deactivating transport protocols during normal platform operations. In the leftmost panel of the RMA GUI, right-clicking on an agent container tree node brings up the popup menu with an *Install a new MTP* and *Uninstall an MTP*.

Choosing *Install a new MTP* a dialog is shown where the user can select the container to install the new MTP on, the fully qualified name of the class implementing the protocol, and (if it is supported by the chosen protocol) the transport address that will be used to contact the new

---

[1] ORBacus is a CORBA 2.3 ORB for C++ and Java. It is available from Object Oriented Concepts, Inc. at http://www.ooc.com. An alternate IIOP MTP for JADE, exploiting ORBacus features, is available in the download area of the JADE web site: http://jade.cselt.it/.

MTP. For example, to install a new IIOP endpoint, using the default JDK 1.3 ORB, one would write `jade.mtp.iiop.MessageTransportProtocol` as the class name and nothing as the address. In order to install a new IIOP endpoint, using the ORBacus based implementation, one would write `orbacus.MessageTransportProtocol` as the class name and (if the endpoint is to be deployed at host `sharon.cselt.it`, on the TCP port 1234, with an object ID `jade`) `corbaloc:iiop:sharon.cselt.it:1234/jade` as the transport address.

Choosing *Uninstall an MTP,* a dialog is shown where the user can select from a list one of the currently installed MTPs and remove it from the platform.

2.5.3 Agent address management

As a consequence of the MTP management described above, during its lifetime a platform, and its agents, can have more than one address and they can be activated and deactivated during the execution of the system. JADE takes care of maintaining consistence within the platform and the addresses in the platform profile, the AMS knowledge base, and in the AID value returned by the method getAID() of the class Agent.

For application-specific purposes, an agent can still decide to choose explicitly a subset of the available addresses to be contacted by the rest of the world. In some cases, the agent could even decide to activate some application specific MTP, that would not belong to the whole platform but only to itself. So, the preferred addresses of an agent are not necessarily the same as the available addresses for its platform. In order to do that, the agent must take care of managing its own copy of agent ID and set the sender of its ACLMessages to its own copy of agent ID rather than the value returned by the method getAID().

2.5.4 Writing new MTPs for JADE

To write a new MTP that can be used by JADE, all that is necessary is to implement a couple of Java interfaces, defined in the `jade.mtp` package. The `MTP` interface models a bi-directional channel that can both send and receive ACL messages (this interface extends the `OutChannel` and `InChannel` interfaces that represent one-way channels). The `TransportAddress` interface is just a simple representation for an URL, allowing separately reading the protocol, host, port and file part.

2.5.4.1 The Basic IIOP MTP

An implementation of the FIPA 2000 IIOP-based transport protocol is included with JADE. This implementation relies on the JDK 1.4 ORB (but can also use the JDK 1.3 ORB, requiring recompilation of the `jade.mtp.iiop` package). This implementation fully supports IOR representations such as `IOR:0000000000000001649444c644f4…`, and does not allow to choose the port number or the object key. These limitations are due to the underlying ORB, and can be solved with other JADE MTPs exploiting more advanced CORBA ORBs. The MTP implementation is contained within the `jade.mtp.iiop.MessageTransportProtocol` class, so this is the name to be used when starting the protocol. Due to the limitation stated above, choosing the address explicitly is not supported.

The default IIOP MTP also supports a limited form of `corbaloc:` addressing: A `corbaloc:` address, generated by some other more advanced ORB and pointing to a different platform, can be used to send ACL messages. Interoperability between a JADE platform using ORBacus and a JADE platform using the JDK 1.3 ORB has been successfully tested. In a first

test, the first platform exported a `corbaloc:` address generated by ORBacus, and then the second platform used that address with the JDK 1.3 ORB to contact the first one. In a second test, the IOR generated by the second platform was converted into a `corbaloc:` URL via the `getURL()` method call in the `IIOPAddress` inner class (a non-public inner class of the `jade.mtp.iiop.MessageTransportProtocol` class); then the first platform used that address to contact the second one.

So, the `corbaloc:` support is almost complete. The only limitation is that it's not possible to export `corbaloc:` addresses with the JDK 1.3 ORB. JADE is able to convert IORs to `corbaloc:` URLs, but the CORBA object key is an arbitrary octet sequence, so that the resulting URL contains forbidden characters that are escaped using '%' and their hexadecimal value. While this conversion complies with CORBA 2.4 and RFC 2396, the resulting URL is just as unreadable as the plain old IOR. The upcoming JDK 1.4 is stated to feature an ORB that complies with the POA and INS specifications, so that it has persistent object references, and natively supports `corbaloc:` and `corbaname:` addresses. It is likely that a more complete IIOP MTP will be provided for the JDK 1.4, when it will be widely available.

2.5.4.2 The ORBacus MTP

A Message Transport Protocol implementation that complies with FIPA and exploits the ORBacus ORB implementation can be download as an add-on from the JADE web site. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this MTP. This MTP fully supports `IOR:`, `corbaloc:` and `corbaname:` addresses.

According to the OMG specifications, three syntaxes are allowed for an IIOP address (all case-insensitive):

```
IIOPAddress ::= "ior:" (HexDigit HexDigit+)
              | "corbaname://" NSHost ":" NSPort "/" NSObjectID
                "#" objectName
              | "corbaloc:" HostName ":" portNumber "/" objectID
```

Notice that, in the third case, BIG_ENDIAN is assumed by default, while in the first and second case, the endianess information is contained within the IOR definition. In the second form, HostName and PortNumber refer to the host where the CORBA Naming Service is running.

2.5.4.3 The HTTP MTP

A Message Transport Protocol implementation that complies to FIPA and uses the HTTP protocol can be download as an add-on from the JADE web site. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this MTP.

2.6 Support for ACL Codec

By default, all ACLMessages are encoded via the String format defined by FIPA. However, at configuration time it is possible to add additional ACLCodecs that can be used by agents on

that container. The command line option *–aclcodec* should be used for this purpose. Agents wishing to send messages with non-default encodings should set the right value in the *aclRepresentation* field of the Envelope.

2.6.1 XML Codec

An XML-based implementation of the ACLCodec can be download from the JADE site as an add-on. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this codec.

2.6.2 Bit Efficient ACL Codec

A bit-efficient implementation of the ACLCodec can be download from the JADE site as an add-on. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this codec. Take care that this codec is available under a different license, not LGPL.

3 AGENT IDENTIFIERS AND SENDING MESSAGES TO REMOTE AGENTS

According to the FIPA specifications, each agent is identified by an Agent Identifier (AID). An Agent Identifier (AID) labels an agent so that it may be distinguished unambiguously within the Agent Universe.

The AID is a structure composed of a number of slots, the most important of which are **name** and **addresses**.

The **name** parameter of an AID is a globally unique identifier that can be used as a unique referring expression of the agent. JADE uses a very simple mechanism to construct this globally unique name by concatenating a user-defined nickname to its home agent platform name (HAP), separated by the '@' character. Therefore, a full valid name in the agent universe, a so-called GUID (Globally Unique Identifier), is peter@kim:1099/JADE   where 'peter' is the agent nickname that was specified at the agent creation time, while 'kim:1099/JADE' is the platform name. Only full valid names should be used within ACLMessages.

The **addresses** slot, instead, should contain a number of transport addresses at which the can be contacted. The syntax of these addresses is just a sequence of URI. When using the default IIOP MTP, the URI for all the local addresses is the IOR printed on stdout. The address slot is defaulted to the addresses of the local agent platform.

4 FAULT-TOLERANCE WITH REPLICATED MAIN CONTAINERS

JADE distributed architecture relies on a special node, named *Main Container*, to coordinate all other nodes and keep together the whole platform. Though most JADE operations are

decentralized, there are some essential features that are supported only by the Main Container. These features are:

- Managing the Container Table (i.e. the set of all the nodes that compose the distributed platform).

- Managing the Global Agent Descriptor Table (i.e. the set of all the agents hosted by the distributed platform, together with their current location).

- Managing the MTP table (i.e. the set of all deployed MTP endpoints, together with their deployment location).

- Hosting the platform AMS agent.

- Hosting the platform Default DF agent.

If the Main Container terminates or otherwise becomes unavailable to the other platform containers, all the above features become unavailable and this severely hampers platform operations.

To keep JADE fully operational even in the event of a failure of the Main-Container, support for Main Container replication has been introduced. Using this support, it is possible to start any number of Main Container nodes (a "*master*" main container actually holding the AMS and a number of "*backup*" main containers), which will arrange themselves in a logical ring so that whenever one of them fails, the others will notice and act accordingly.

Ordinary containers will then be able to connect to the platform through any of the active Main Container nodes; the different copies will evolve together using cross-notification.

As the following figure shows, without Main Container replication JADE platform has a star topology, end enabling Main Container replication turns the topology into a ring of stars.
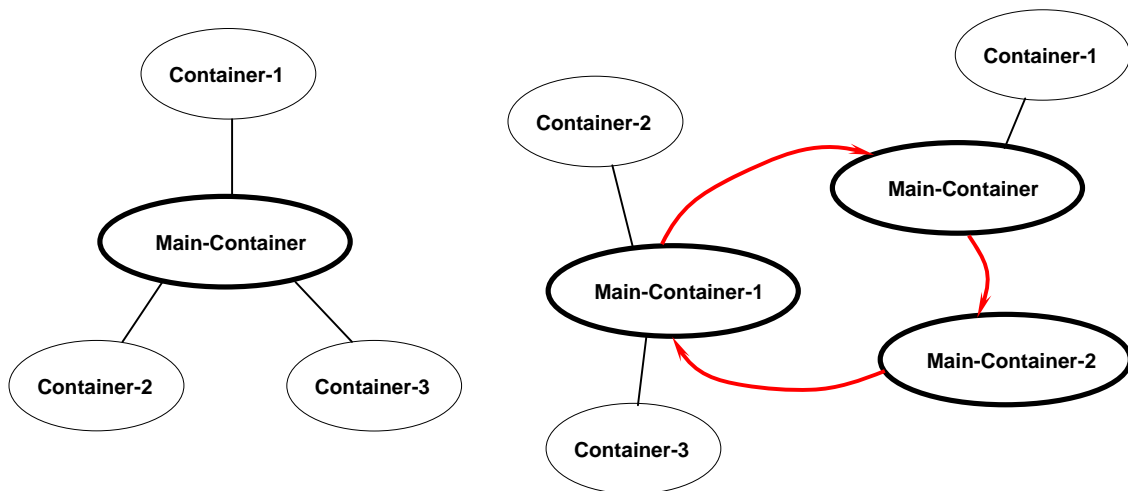
In the fault-tolerant configuration shown on the right part of the figure, three Main Container nodes are arranged in a ring, and each node is monitoring its neighbor: if the node `Main-Container-1` fails, the node `Main-Container-2` will notice and inform all the other Main Container nodes (in this case just the `Main-Container` one). Then, a smaller ring will be rebuilt (with just two elements in the current example).

The figure also shows that peripheral containers can be arbitrarily spread among the available Main Container nodes. Any single peripheral container is connected to exactly one node and in absence of failures it is completely unaware of all the other copies. When a Main

Container node fails, there will generally be some orphaned peripheral containers (in the current example, supposing a `Main-Container-1` failure, `Container-3` will become an orphan). When an orphan container detects that its Main Container node isn't available anymore, it attaches itself to another node: for this to succeed, a peripheral container must know the list of all the Main Container nodes present in the platform.

JADE supports two policies in distributing the Main Container node list to peripheral containers. A first option is to activate the Address-Notification service on all Main Container nodes and on the peripheral containers. This service will detect additions and removals to the Main Container nodes ring and update the address lists of all peripheral containers involved.

A second option is to pass the address list to a starting peripheral container with `-smaddrs` command line argument. This approach avoids generating notification traffic towards peripheral containers but assumes a fixed list of Main Container nodes, which is known beforehand.

4.1 Starting a Fault Tolerant Platform from the Command Line

A variety of fault-tolerant platform configurations can be obtained by different combinations of a few command line options; this section will show some examples, mapping a sequence of commands into the graphical representation of the platform configuration they create. In the following, the various commands will supposedly be written at a command prompt that comprises the host name: that is, a `user@host1:` command prompt means that the `user` login name was used to connect to the `host1` machine. Commands typed by the user will be shown in bold face as usual.

4.1.1 Step 1: Two Main Container nodes on two host machines.

```
user@host1: java jade.Boot -name Hydra -services
jade.core.replication.MainReplicationService;jade.core.replication.AddressNotificationServ
ice
```

The command above starts a master Main Container node on machine `host1` and activates the Main-Replication and Address-Notification services on it (notice that such a command line does not activate the Agent-Mobility and Notification services; refer to Section 2.3.2 for further details). The `-name` option assigns a name ("`Hydra`" in this case) to the platform constituted by the newly started main container. When starting more than one main container in the same platform (as we are doing in this example) to create a fault-tolerant topology it is necessary to specify the `-name` option for each[2] main container so that they all see the same platform name. Even if not required it is suggested to specify the `-name` option (with the same value) also for peripheral containers.
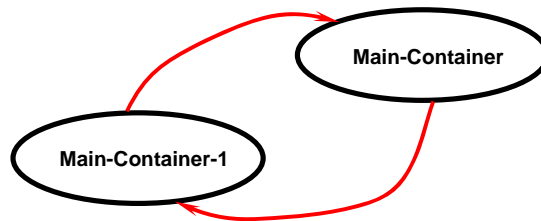
```
user@host2: java jade.Boot -backupmain -local-port 1234 -host host1 -port 1099 -name Hydra
-services
jade.core.replication.MainReplicationService;jade.core.replication.AddressNotificationServ
ice
```

The command above, beyond activating the two services needed by a fault-tolerant platform, uses the `-backupmain` option to specify that the newly started node is a Main Container, but does not make a new platform on its own. Rather, this new node is to join an existing platform that is specified by the `-host`, `-port` and `-name` options. In our example, these options point exactly at the Main Container that was started with the first command line. Moreover the `-local-port`

---

[2] The `-name` option can be omitted when launching the first main container (the master). In this case the platform name is automatically set to `host:port/JADE`. Other (backup) main containers must be launched specifying that name.

option specifies the port that new containers will have to indicate to connect to this main container.

The graphical representation of the JADE platform follows.

4.1.2 Step 2: Adding peripheral containers

Regardless their activation order, the two Main Container nodes started in the previous step are now absolutely interchangeable. The platform built so far is a single, redundant platform named `Hydra`, exposing two Service Manager addresses (namely, `host1:1099` and `host2:1234`) for peripheral containers to join. In this step we will start two such containers, one for each address (and thus one for each active Main Container node).
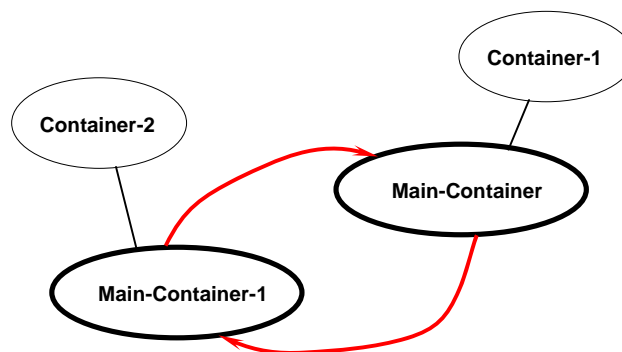
```
user@host3: java jade.Boot –container –host host1 -port 1099 -name Hydra –services
jade.core.replication.AddressNotificationService
```

The above command uses the `-container` option to launch a peripheral container and activates the Address-Notification service on it. The container will join the `Hydra` platform using the host `host1` and the port `1099`, thus connecting to the `Main-Container` node.

```
user@host4: java jade.Boot –container –host host2 –port 1234 –name Hydra –smaddrs
host1:1099;host2:1234
```

This second command is similar to the first one, but here we chose not to activate the Address-Notification service; instead, we used the `-smaddrs` option to provide the list of the available Service Manager addresses. Since we want this container to connect to the `Main-Container-1` node, we use suitable `–host` and `-port` options.

The shape of the agent platform after this step is shown in the following figure.

4.1.3 Step 3: Completing the platform.

In this final step we start one more Main Container node and one more peripheral container node, so that we reach the platform topology used at the beginning of this section as an example of fault-tolerant platform.

```
user@host5: java jade.Boot –backupmain –local-port 1099 –host host1 –port 1099 –name Hydra
–services
jade.core.replication.MainReplicationService;jade.core.replication.AddressNotificationServ
ice
```

The above command starts a third Main Container node, exporting a Service Manager address on host `host5`, port 1099.

```
user@host6: java jade.Boot –container –host host2 –port 1234 -name Hydra –services
jade.core.replication.AddressNotificationService
```

The above command starts a peripheral container with the Address-Notification service active, joining the platform through the `Main-Container-1` node. With this last command we obtained the complete platform topology mentioned earlier.

4.2 Surviving Main Container failures.

After explaining how to set up a redundant agent platform in the previous subsections, we now describe how such a platform can recover and reconfigure itself after a Main Container node failure, continuing normal operations. For the purpose of this example, we will suppose that all application agents are hosted by the three peripheral containers, and that one of the Main Container nodes will terminate abruptly: let that be the `Main-Container-1` node.

The recovery process can be observed if an RMA agent is started on one of the peripheral containers. As soon as the `Main-Container-1` node terminates, the `Main-Container-2` node will notice and inform the `Main-Container` node. A new, tighter Main Container ring will result (composed of the two surviving nodes). Nothing happens to the `Container-1` node, while the other two peripheral containers will reconnect themselves to another Main Container node as soon as they need it. After this failure, the new platform topology could look like this.
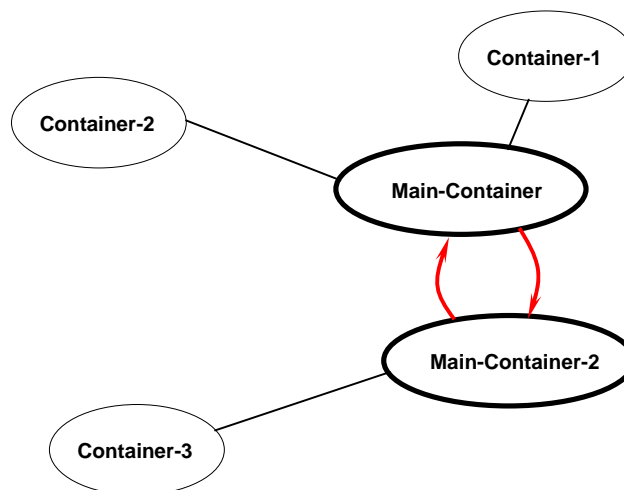
All Main Container nodes are replicas of each other, with one single difference. Only one of the replicas (the master main container) appears to actually host the platform AMS and Default DF

agents: this one is the first replica in the ring, which is `Main-Container` node in our case. The previous crash of `Main-Container-1` node left the AMS and Default DF untouched, but let us now consider the case of `Main-Container` node failure.

The course of events is the same, with `Main-Container-2` node realizing the failure and the three peripheral containers reconnecting to it when needed. However, this time the `Main-Container-2` node realizes that the failed node was hosting the platform system agents and proceeds to activate its own replicas of the AMS and Default DF (each Main Container node has a dormant replica of the two, which is kept updated just like the other platform-level data structures). From the perspective of an RMA agent living on e.g. `Container-3` node, the only difference is that `Main-Container` has disappeared from the platform and `ams` and `df` agents have now moved on `Main-Container-2`.

5 PERSISTENT DELIVERY OF ACL MESSAGES

JADE allows the platform administrator to define a buffering, storage and retry strategy for undelivered ACL messages. When the receiver agent is not found in the platform, the ACL message delivery fails and a suitable `FAILURE` message is sent back to the originator, in compliance with the FIPA specifications.

Using the `Persistent-Delivery` kernel service, it is possible to avoid sending back the `FAILURE` notification, and buffer the undelivered message instead. Messages are matched against a user-defined filter and can simply be buffered in memory, or they can be stored within the file system. Different containers in the platform can define different filters and storage methods.

5.1 Defining a filter for ACL messages

In order to provide application-specific logic to the Persistent-Delivery service, a Java class must be defined and compiled. This class has to implement the `PersistentDeliveryFilter` interface, defined in the `jade.core.messaging` package and reported below.

```
public interface PersistentDeliveryFilter {

    static final long NOW = 0;
    static final long NEVER = -1;


    long delayBeforeExpiration(ACLMessage msg);

}
```

As the code above shows, there is just a single method in the interface. Whenever an ACL message fails to be delivered to one of its receivers, the persistent delivery filters installed on all the platform containers are given a chance to claim the message for buffering. The `delayBeforeExpiration()` method of each filter is called, and the undelivered ACL message is passed to it. If the method call returns `NOW`, the message is considered unclaimed by the filter, and another filter (on a different container) is invoked. This process goes on until a filter claims the message (by returning something other than `NOW`) or all installed filters have been invoked (in the second case, the `FAILURE` notification is sent to the message sender).

If a filter returns a different value from `NOW`, it is considered as a time delay in milliseconds. The ACL message will be buffered on the node where the filter is deployed, and a delivery attempt will be periodically made. If the message is still undelivered after the delay returned by the filter, the delivery process will abort and the `FAILURE` will be sent back to the message sender.

The following code shows a sample implementation of the `delayBeforeExpiration()` method. In this example, all the messages belonging to a news delivery application (identified by their `:ontology` slot) are claimed. If a claimed message defines a `:reply-by` slot, a suitable delay is generated, otherwise the message is buffered indefinitely.

```
public long delayBeforeExpiration(ACLMessage msg) {

    if(msg.getOntology().equals("News-Ontology")) {
        Date d = msg.getReplyByDate();
        if(d != null) {
            long delay = d.getTime() - System.currentTimeMillis();
            return (delay > 0) ? delay : 0;
        }
        else {
            return NEVER;
        }
    }
    else {
        return NOW;
    }
}
```

5.2 Installing and configuring the Persistent-Delivery service

The Persistent-Delivery kernel service can be installed on a container at start-up time, just like the other JADE services. A set of profile properties is defined to allow flexible configuration of the service.

- The `persistent-delivery-filter` property names the application-defined class (a subtype of `jade.core.messaging.PersistentDeliveryFilter` interface) that will be dynamically loaded and installed as the message filter on this container.

- The `persistent-delivery-sendfailureperiod` property defines how often the service will try to deliver again ACL messages, checking their expiration in the process. This property is expressed in milliseconds and its default value is 60000 (one minute).

- The `persistent-delivery-storagemethod` property specifies how buffered ACL messages are to be persisted. By default, messages are kept in memory and no persistent storage is used. A filesystem-based persistent storage is provided, which can be selected giving the `file` value to this profile property.

- The `persistent-delivery-basedir` property is used by the `file` persistent storage method. The directories and files holding the persisted buffered messages are created as children of the directory given as value of this property. If this property is not used, by default a subdirectory of the current directory is used, named `PersistentDeliveryStore`.

If the `file` persistent storage method is used, a directory tree in the local filesystem is used to hold the buffered messages. The directory tree is organized as follows:

A *base directory* is created according to the value of the `persistent-delivery-basedir` profile property. Undelivered ACL messages are encapsulated in *delivery items* objects. A delivery item contains:

- The ACL message.

25

- The AID of the intended receiver.
- The due date within which the message is to be delivered (calculated using the delay provided by the persistent delivery filter which claimed the message).

The intended receiver AID is hashed to a numeric value, which is used to create a subdirectory for the base directory. Thus, all undelivered messages for a given AID will end up in the same directory.

The `<ACL Message; Receiver AID>` pair is hashed to a numeric value, which is used to create a file within the aforementioned directory. That file contains, orderly:

- The number of message copies currently buffered (if the same message is sent many times, it consistently hashes to the same file).
- The absolute due date for message delivery, or the keyword `FOREVER`.
- The AID of the intended receiver, in SL format.
- The whole ACL message, in S-expression format.

When an undelivered ACL message is stored, a suitable file with the above content is created, and as soon as the message is delivered or the `FAILURE` notification is sent back the file is deleted. When the service starts up, it looks for delivery items from previous platform executions and loads them in memory.

5.3 Command-line examples

The Persistent-Delivery service is not activated by default, so an explicit `-services` command-line parameter is needed. The following example starts a container with the Persistent-Delivery service on it, a `MyFilter` message filter class and no persistent storage of undelivered messages.

```
user@host: java jade.Boot –persistent-delivery-filter MyFilter -services
jade.core.messaging.PersistentDeliveryService
```

The following command line, instead, configures the Persistent-Delivery service so that it uses a `app.news.NewsMessage` filter class and stores undelivered messages under the `/usr/local/news` directory.

```
user@host: java jade.Boot –persistent-delivery-filter app.news.NewsMessage -services
jade.core.messaging.PersistentDeliveryService –persistent-delivery-storagemethod file
–persistent-delivery-basedir /usr/local/news
```

6 GRAPHICAL USER INTERFACE TO MANAGE AND MONITOR THE AP ACTIVITY

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

6.1 Remote Monitoring Agent

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

An RMA is a Java object, instance of the class `jade.tools.rma.rma` and can be launched from the command line as an ordinary agent (i.e. with the command `java`

jade.Boot myConsole:jade.tools.rma.rma), or by supplying the '-gui' option the command line parameters (i.e. with the command java jade.Boot –gui).

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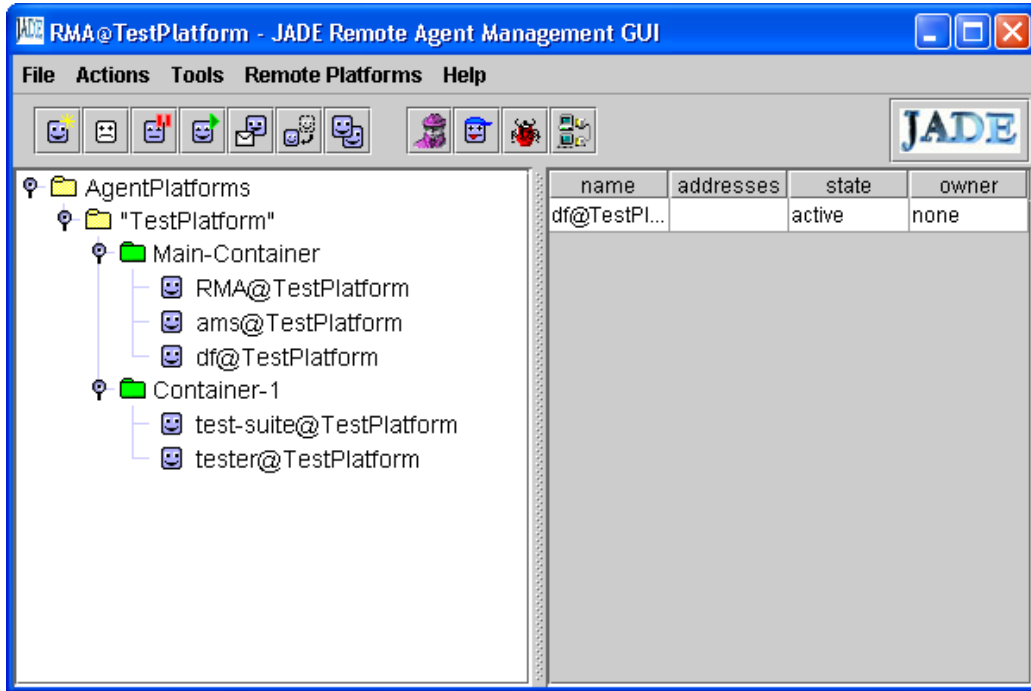

Figure 2 Snapshot of the RMA GUI

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

♦ File menu:

This menu contains the general commands to the RMA.

♦ Close RMA Agent

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

♦ Exit this Container

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

♦ Shut down Agent Platform

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

♦ Actions menu:

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

♦ Start New Agent

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

♦ Kill Selected Items

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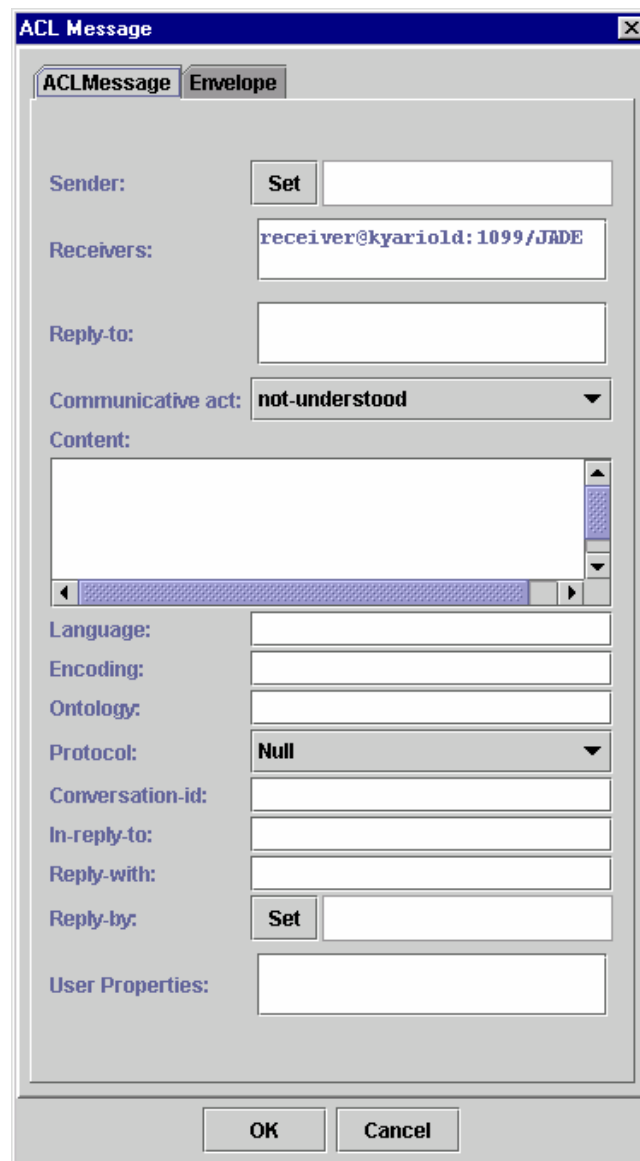
♦ Suspend Selected Agents

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

♦ Resume Selected Agents

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

♦ Send Custom Message to Selected Agents

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

♦ Migrate Agent

This action allows to migrate an agent. When the user selects this menu item, a special dialog is displayed in which the user must specify the container of the platform where the selected agent must migrate. Not all the agents can migrate because of lack of serialization support in their implementation. In this case the user can press the cancel button of this dialog.

♦ Clone Agent

This action allows to clone a selected agent. When the user selects this menu item a dialog is displayed in which the user must write the new name of the agent and the container where the new agent will start.

♦ Tools menu:

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

♦ RemotePlatforms menu:

This menu allows controlling some remote platforms that comply with the FIPA specifications. Notice that these remote platforms can even be non-JADE platforms.

♦ Add Remote Platform via AMS AID

This action allows getting the description (called APDescription in FIPA terminology) of a remote Agent Platform via the remote AMS. The user is requested to insert the AID of the remote AMS and the remote platform is then added to the tree showed in the RMA GUI.

♦ Add Remote Platform via URL

This action allows getting the description (called APDescription in FIPA terminology) of a remote Agent Platform via a URL. The content of the URL must be the stringified APDescription, as specified by FIPA. The user is requested to insert the URL that contains the remote APDescription and the remote platform is then added to the tree showed in the RMA GUI.

♦ View APDescription

To view the AP Description of a selected platform.

♦ Refresh APDescription

This action asks the remote AMS for the APDescription and refresh the old one.

♦ Remove Remote Platform

This action permits to remove from the GUI the selected remote platform.

♦ Refresh Agent List

This action performs a search with the AMS of the Remote Platform and the full list of agents belonging to the remote platform are then displayed in the tree.

6.2 DummyAgent

The DummyAgent tool allows users to interact with JADE agents in a custom way. The GUI allows composing and sending ACL messages and maintains a list of all ACL messages sent and received. This list can be examined by the user and each message can be viewed in detail or even edited. Furthermore, the message list can be saved to disk and retrieved later. Many instances of the DummyAgent can be started as and where required.

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

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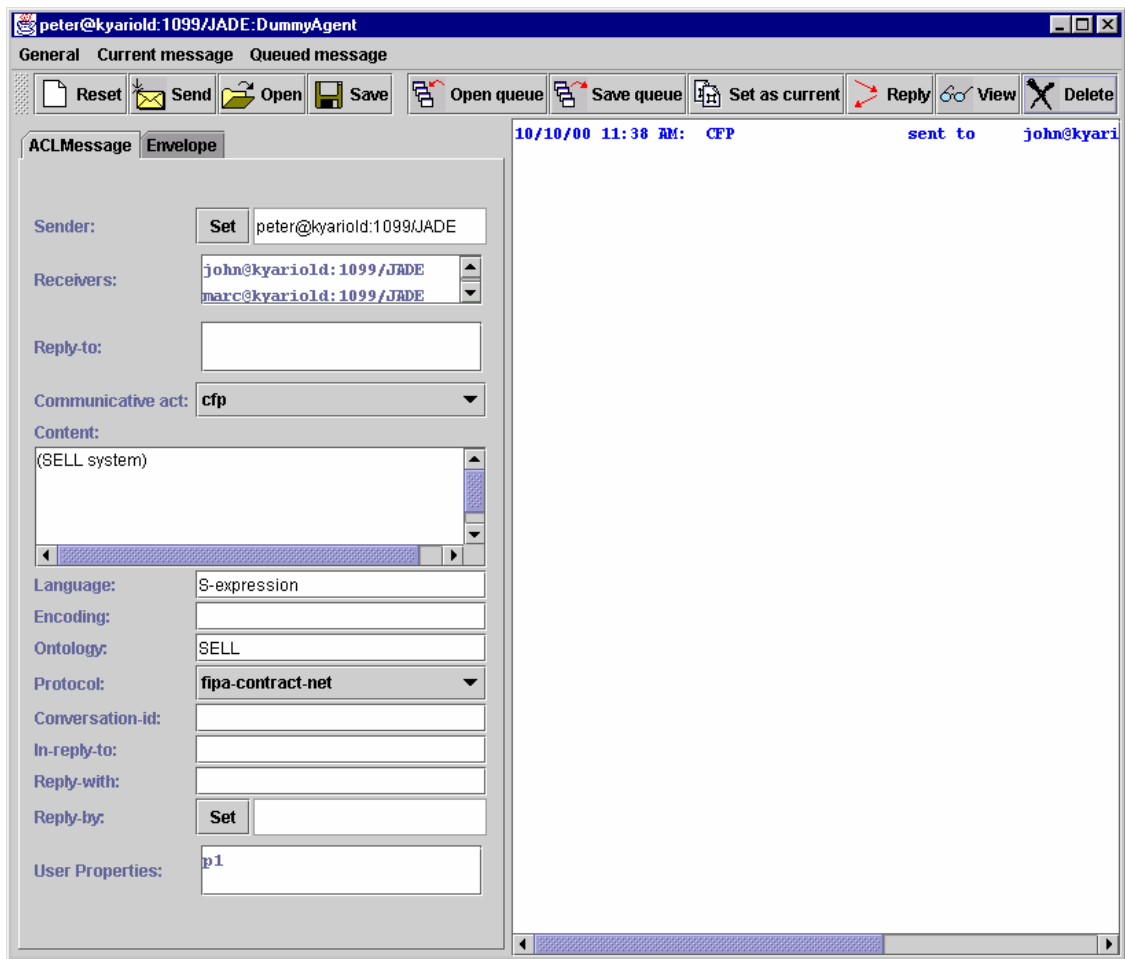

Figure 3 Snapshot of the DummyAgent GUI

6.3 DF GUI

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

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

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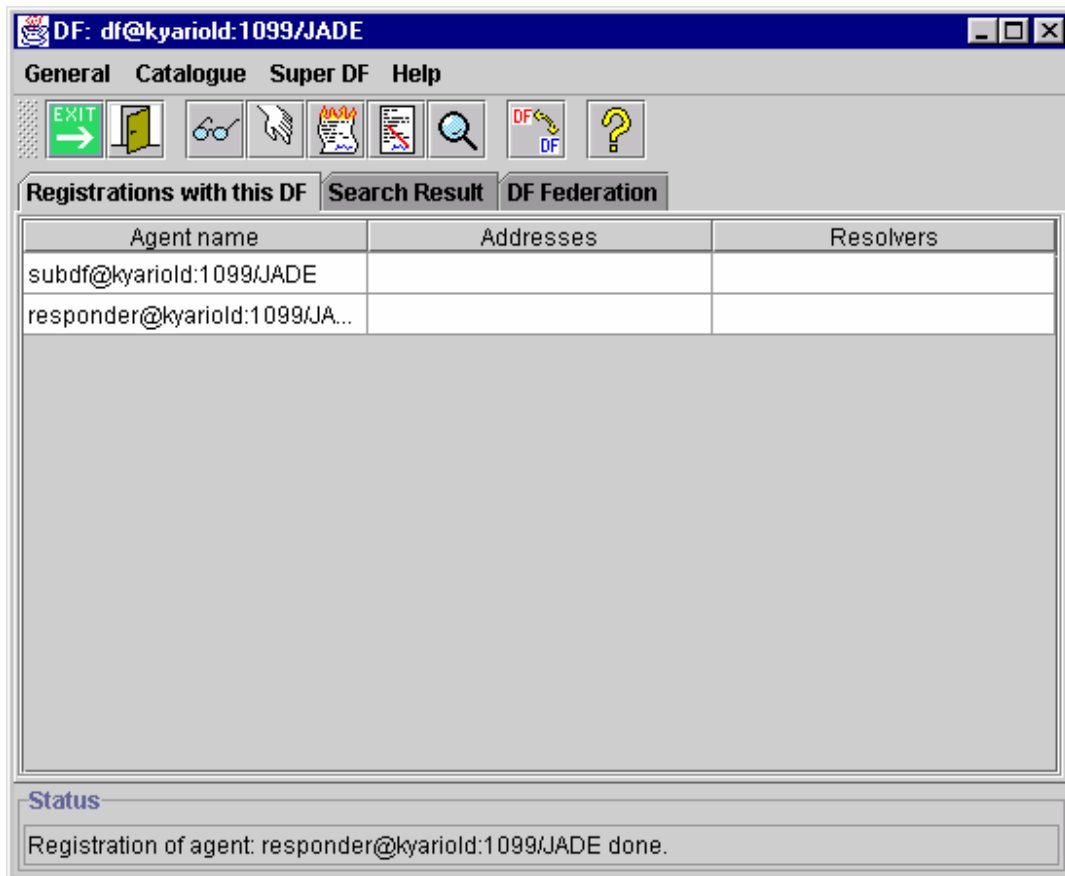

Figure 3 – Snapshot of the GUI of the DF

6.4 Sniffer Agent

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

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

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

```
java jade.Boot sniffer:jade.tools.sniffer.Sniffer
```
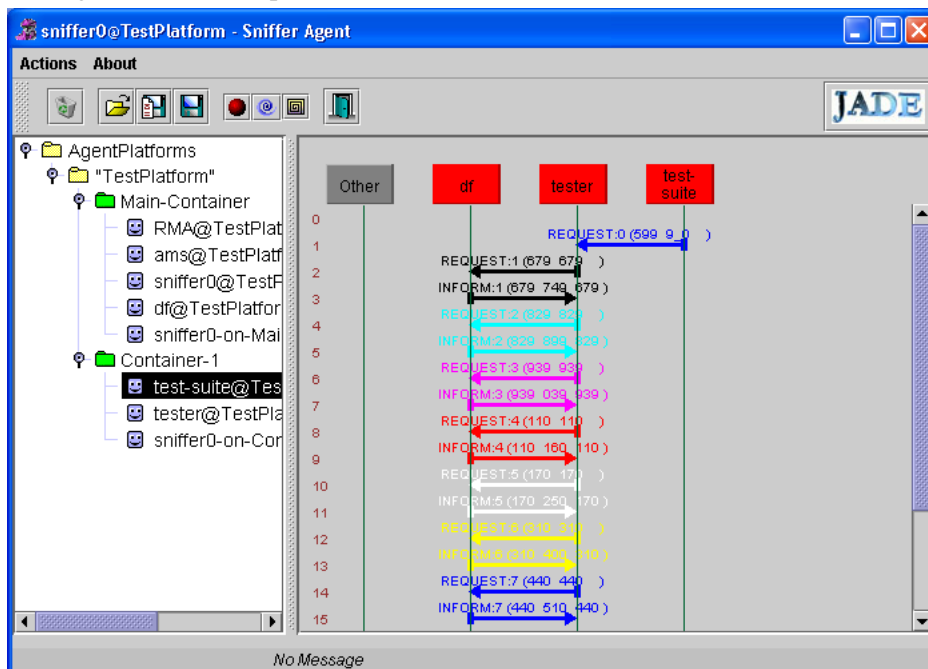
The figure shows a snapshot of the GUI.

Figure 4 - Snapshot of the sniffer agent GUI

At start up, the sniffer subscribes itself with the platform in order to be informed every time an agent is born or dies, as well as a container is created or deleted.

A properties file may be used to control different sniffer properties. These optional properties are as follows:

- `preload` - A list of preload descriptions seperated by a semi-colon. Each description consists of an agent name match string and optional list of performatives each seperated by a space. If there is no @ in the agent name, it assumes the current HAP for it. If the performative list is not present, then the sniffer will display all messages; otherwise, only those messages that have a matching performative mentioned will be displayed.

    Examples:

    ```
    preload=da0;da1 inform propose
    ```

33

```
preload=agent?? inform
preload=*
```

- `clip` - A list of agent name prefixes separated by a semi-colon which will be removed when showing the agent's name in the agent box. This is helpful to eliminate common agent prefixes.

  Example:

  ```
  clip=com.hp.palo-alto.;helper.
  ```

The property file called *'sniffer.properties'* is looked for in the current directory, and if not found, the agent looks in the parent directory and continues this until the file is either found or there isn't a parent directory.

The original implementation processed a file called *'sniffer.inf'* file. For backward compatability this has been preserved but its usage should be converted to use the new *.properties* file. The format of the *.inf* file is a set of lines, where each line contains an agent name to be sniffer and optional list of performatives.

Example:

```
da0
da1 inform propose
```

Notes:

If a message is one that is to be ignored, then it is dropped totally. If you look at the sniffer dump of messages, it will not be there.

The same functionality is provided via command line arguments for the Sniffer Agent.

Example:

```
java jade.Boot
    Sniffer:jade.tools.sniffer.Sniffer(df* ; RMA inform)
```

6.5 Introspector Agent

This tool allows to monitor and control the life-cycle of a running agent and its exchanged messages, both the queue of sent and received messages. It allows also to monitor the queue of behaviours, including executing them step-by-step.
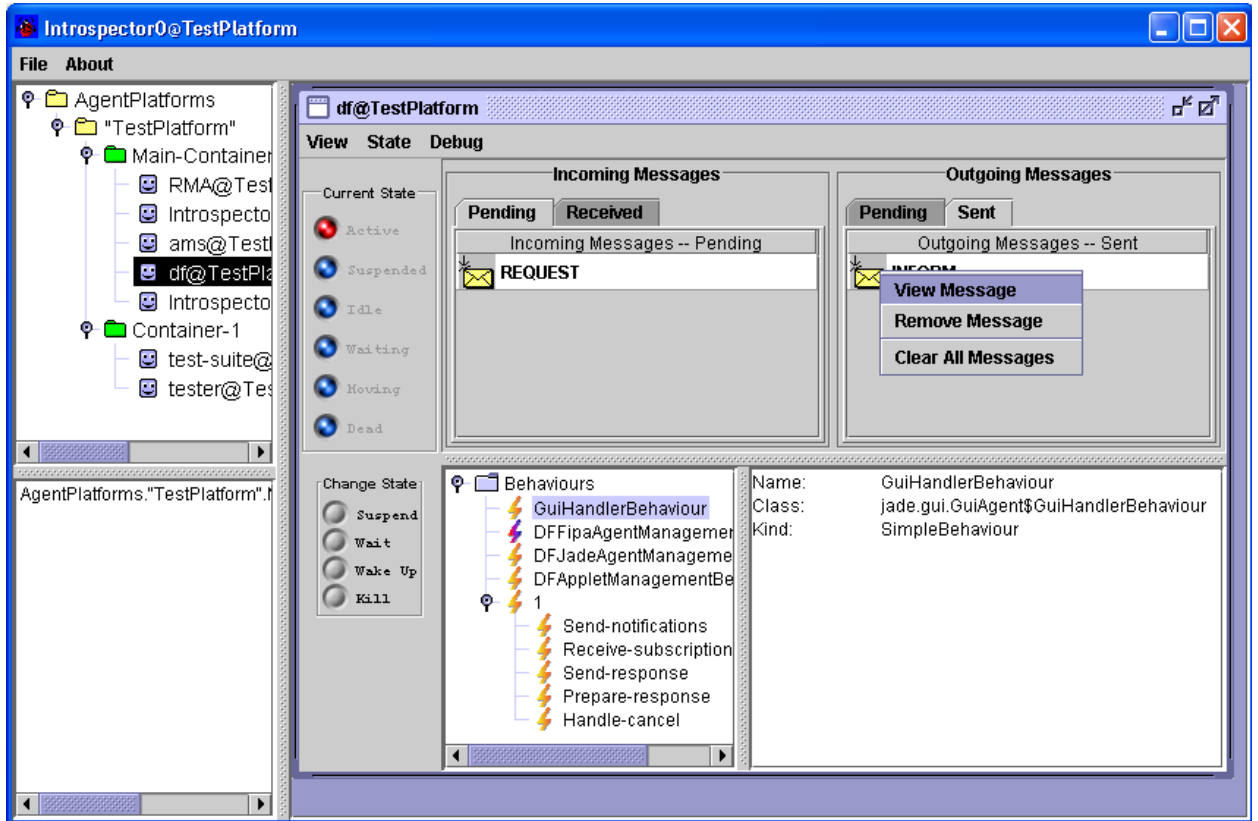

Figure 4 - Snapshot of the Introspector Agent GUI

Agents can be passed to the Introspector Agent in the same way as to the Sniffer Agent via the command line, or via a configuration file. The specification of performative filters, e.g. inform, agree, etc. is not supported by the Introspector Agent.

7 LIST OF ACRONYMS AND ABBREVIATED TERMS

ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management Service. According to the FIPA architecture, this is the agent that is responsible for managing the platform and providing the white-page service.
AP	Agent Platform
API	Application Programming Interface
DF	Directory Facilitator. According to the FIPA architecture, this is the agent that provides the yellow-page service.
EBNF	Extended Backus-Naur Form
FIPA	Foundation for Intelligent Physical Agents
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HAP	Home Agent Platform
HTML	Hyper Text Markup Language
HTTP	Hypertext Transmission Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
INS	
IOR	Interoperable Object Reference
JADE	Java Agent DEvelopment Framework
JDK	Java Development Kit
JVM	Java Virtual Machine
LGPL	Lesser GNU Public License
MTP	Message Transport Protocol. According to the FIPA architecture, this component is responsible for handling communication with external platforms and agents.
ORB	Object Request Broker
POA	Portable Object Adapter
RMA	Remote Monitoring Agent. In the JADE platform, this type of agent provides a graphical console to monitor and control the platform and, in particular, the life-cycle of its agents.
RMI	Remote Method Invocation
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language