JADE – A FIPA-compliant agent framework

Fabio Bellifemine
CSELT S.p.A.
Via G. Reiss Romoli, 274,
10148,
Torino Italy
Tel. +39 011 2286175
Fax. +39 011 2286175

bellifemine@cselt.it

Agostino Poggi
DII – University of Parma
Parco Area delle Scienze, 181A
Tel. +39 521 905728
Fax. +39 0521 905723

poggi@ce.unipr.it

Giovanni Rimassa
DII – University of Parma
Parco Area delle Scienze, 181A
Tel. +39 521 905728
Fax. +39 0521 905723

rimassa@ce.unipr.it

Abstract

JADE is a software framework to develop agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. The goal is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. JADE can then be considered an agent middle-ware that implements an Agent Platform and a development framework. It deals with all those aspects that are not peculiar of the agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle. This paper presents the JADE software describing its intended uses, as well as being a walkthrough of JADE internal architecture. The main architectural issues are discussed, and the major design decisions are outlined.

1. INTRODUCTION

The growth in networked information resources requires information systems that can be distributed on a network and interoperate with other systems. Such systems cannot be easily realized with traditional software technologies because of the limits of these technologies in coping with distribution and interoperability. The agent-based technologies seem be a promising answer to facilitate the realization of such systems because they were invented to cope with distribution and interoperability [6].

Agent-based technologies are still immature and few truly agent-based systems have been realized. Agent-based technologies cannot realize their full potential, and will not become widespread, until standards to support agent interoperability are available and used by agent developers and adequate environments for the development of agent systems are available.

Several researchers are working towards the standardization of agent technologies (see, for example, the work done by Knowledge Sharing Effort [10], OMG [9] and FIPA [5]) and in the realization of development environments to build agent systems (see, for example, RETSINA [13], MOLE [11] and ZEUS [8]). Such development environments provide some predefined agent models and tools to make easy the development of systems. Moreover, some of them try to allow interoperability with other agent systems through the use of a well-known agent communication language, that is, KQML [3]. However, the use of a common communication language is not enough to easily support interoperability between different agent systems. The standardization work of FIPA is in the direction to allow an easy interoperability between agent systems, because FIPA, beyond the agent communication language, specifies also the key agents necessary for the management of an agent system, the ontology necessary for the interaction between systems, and it defines also the transport level of the protocols.

In this paper, we present JADE (Java Agent DEvelopment Framework) that is a software framework to develop agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. The next section introduces FIPA specifications. Section three describes JADE main features and, in particular, the architecture of the agent platform, the communication subsystem and the agent execution model. Finally, section four concludes with a brief discussion about the current implementation of JADE, its current use to develop applications and to test the interoperability with other agent platforms, and the new features that will be added in the future.

2.   FIPA SPECIFICATIONS

The Foundation for Intelligent Physical Agents (FIPA) [5] is an international non-profit association of companies and organizations sharing the effort to produce specifications of generic agent technologies. FIPA is envisaged not just as a technology for one application but as generic technologies for different application areas, and not just as independent technologies but as a set of basic technologies that can be integrated by developers to make complex systems with a high degree of interoperability.

FIPA is based on two main assumptions. The first is that the time to reach consensus and to complete the standard should not be long, and, mainly, it should not act as a brake on progress rather than an enabler, before industries make commitments. The second is that only the external behavior of system components should be specified, leaving implementation details and internal architectures to agent developers. In fact, the internal architecture of JADE is proprietary even if it complies with the interfaces specified by FIPA.

The first output documents of FIPA, called FIPA97 specifications, specify the normative rules that allow a society of agents to inter-operate, that is effectively exist, operate and be managed. First of all they describe the reference model of an agent platform, as shown in Figure 1. Basically, it identifies the roles of some key agents necessary for the management of the platform, and specifies the agent management content language and ontology. Three key mandatory roles were identified into an agent platform. The Agent Management System (AMS) is the agent that exerts supervisory control over access to and use of the platform; it is responsible for authentication of resident agents and control of registrations. The Agent Communication Channel (ACC) is the agent that provides the path for basic contact between agents inside and outside the platform; it is the default communication method which offers a reliable, orderly and accurate message routine service; it must also support IIOP for interoperability between different agent platforms. The Directory Facilitator (DF) is the agent that provides a yellow page service to the agent platform. Notice that no restriction is given to the actual technology used for the platform implementation: e-mail based platform, CORBA based, Java multi-thread applications, … could all be FIPA compliant implementations.

Of course, the standard specifies also the Agent Communication Language (ACL). Agent communication is based on message passing, where agents communicate by formulating and sending individual messages to each other. The FIPA ACL specifies a standard message language by setting out the encoding, semantics and pragmatics of the messages. The standard does not set out a specific mechanism for the internal transportation of messages. Instead, since different agents might run on different platforms and use different networking technologies, FIPA specifies that the messages transported between platforms should be encoded in a textual form. It is assumed that the agent has some means of transmitting this textual form. The syntax of the ACL is very close to the widely used communication language KQML. However, despite syntactic similarity, there are fundamental differences between KQML and ACL, the most evident being the existence of a formal semantics for ACL which should eliminate any ambiguity and confusion from the usage of the language.

The standard supports common forms of inter-agent conversations through the specification of *interaction protocols*, which are patterns of messages exchanged by two or more agents. Such protocols include range from simple query-request protocols, to the well-known contract net negotiation protocol and English and Dutch auctions.

Other parts of the FIPA standard specify other aspects, in particular the agent-software integration, agent mobility and security, ontology service, and the Human-Agent Communication. However, they have not yet been considered into the JADE implementation. The interested reader should refer directly to the FIPA Web page [5].
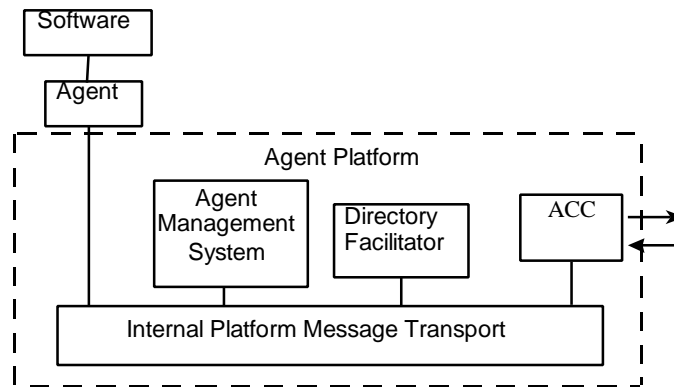
Figure 1 – FIPA reference model of an Agent Platform

3. JADE

JADE (Java Agent DEvelopment Framework) is a software framework to make easier the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. The goal of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. To achieve such a goal, JADE offers the following list of features 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 agents are automatically activated at the agent platform start-up;

- distributed agent platform. The agent platform can be split on several hosts (provided that there is no firewall between them). Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as one Java thread and Java events are used for effective and light-weight communication between agents on the same host. Parallel tasks can be still executed by one agent, and JADE schedules these tasks in a more efficient (and even simpler for the skilled programmer) way than the Java Virtual Machine does for threads;

- a number of FIPA-compliant DFs (Directory Facilitator) can be started at run time in order to implement multi-domain applications, where the notion of domain is a logical one as described in FIPA97 Part 1;

- programming interface to simplify registration of agent services with one, or more, domains (i.e. DF);

- transport mechanism and interface to send/receive messages to/from other agents;

- FIPA97-compliant IIOP protocol to connect different agent platforms;

- light-weight 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 sender or receiver do not belong to the same platform, the

message is automatically converted to /from the FIPA compliant string format. In this way, this conversion is hidden to the agent implementers that only need to deal with the same class of Java object;

- library of FIPA interaction protocols ready to be used;

- automatic registration 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 platforms from the same agent. The activity of each platform can be monitored and logged.

3.1. Architecture of the Agent Platform

The JADE Agent Platform complies with FIPA97 specifications and includes all those mandatory agents that manage the platform, that is the ACC, the AMS, and the DF. All agent communication is performed through message passing, where FIPA ACL is the language to represent messages.
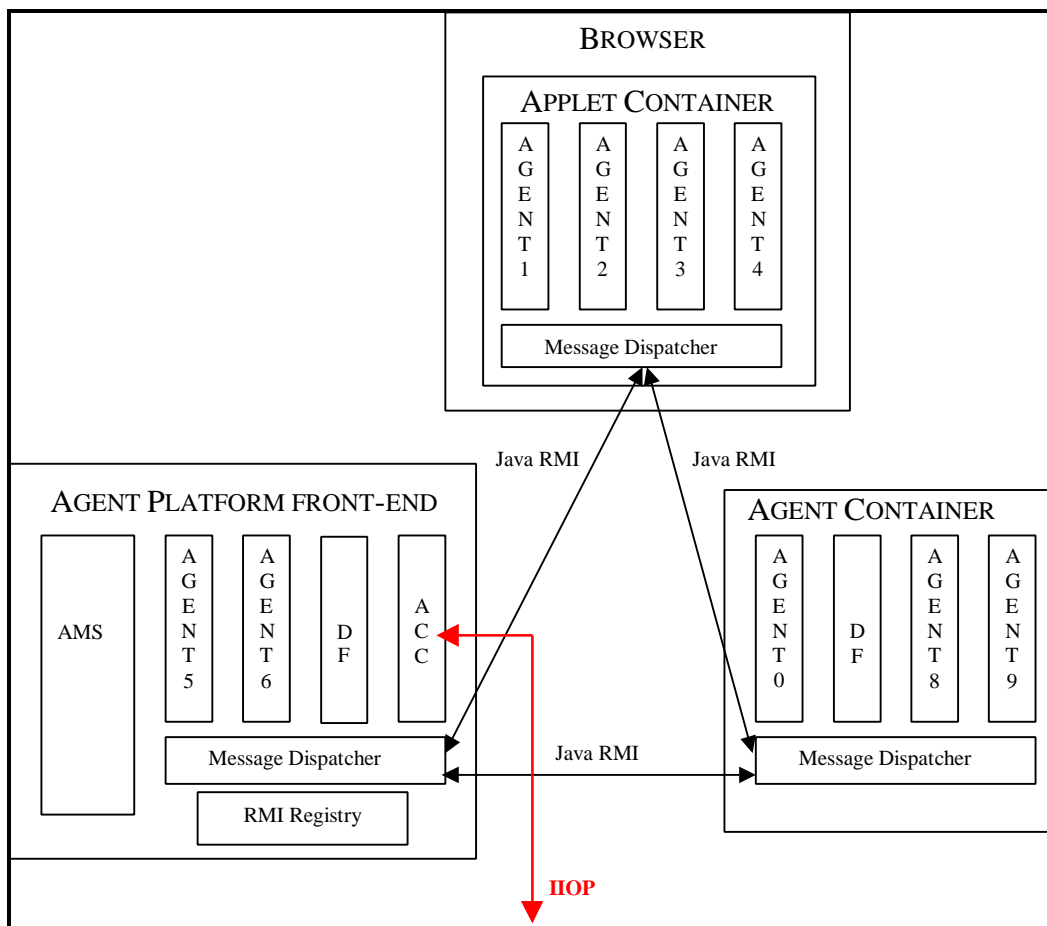

Figure 2 - Software Architecture of one JADE Agent Platform

The software architecture is based on the coexistence of several Java Virtual Machines (VM) and communication relies on Java RMI (Remote Method Invocation) between different VMs and event signaling within a single VM. Each VM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. In principle, the architecture allows also several VMs to be executed on the same host; however, this is discouraged because of the increase in overhead and the lack of whatever benefit. Each agent container is a multithreaded execution environment composed of one thread for every

agent plus system threads spawned by RMI runtime system for message dispatching. A special container plays the front-end role, running management agents and representing the whole platform to the outside world. A complete Agent Platform (AP) is then composed of several agent containers as shown in Figure 2. Distribution of containers across a computer network is allowed, provided that RMI communication between their hosts is preserved. A special light-weight container is being implemented for the execution of agents within a Web browser.

Figure 3 - View of a sample Agent Description registered with the DF

Each *Agent Container* is an RMI server object that locally manages a set of agents. It controls the life cycle of agents by creating, suspending, resuming and killing them. Besides, it deals with all the communication aspects by dispatching incoming ACL messages, routing them according to the destination field (*:receiver*) and putting them into private agent message queues; for outgoing messages, instead, the *Agent Container* maintains enough information to look up receiver agent location and choose a suitable transport to forward the ACL message.

The agent platform provides a Graphical User Interface (GUI) for the remote management, monitoring and controlling of the status of agents, allowing, for example, to stop and restart agents. The GUI allows also to create and start the execution of an agent on a remote host, provided that an agent container is already running. The GUI itself has been implemented as an agent, called RMA (Remote Monitoring Agent). All the communication between agents and this GUI and all the communication between this GUI and the AMS is done through ACL via an ad-hoc extension of the Fipa-agent-management ontology. This extension is going to be submitted to FIPA for standardization consideration. If it were accepted, the same RMA agent might be used also to control other agent platforms, included non-JADE platforms.

Figure 3 is the description of an agent registered with the DF, as shown by the JADE GUI. Figure 4 is instead the graphical representation of the platforms, the componing agent containers, and the agents running on each container, that is mostly the knowledge captured by the AMS agent.
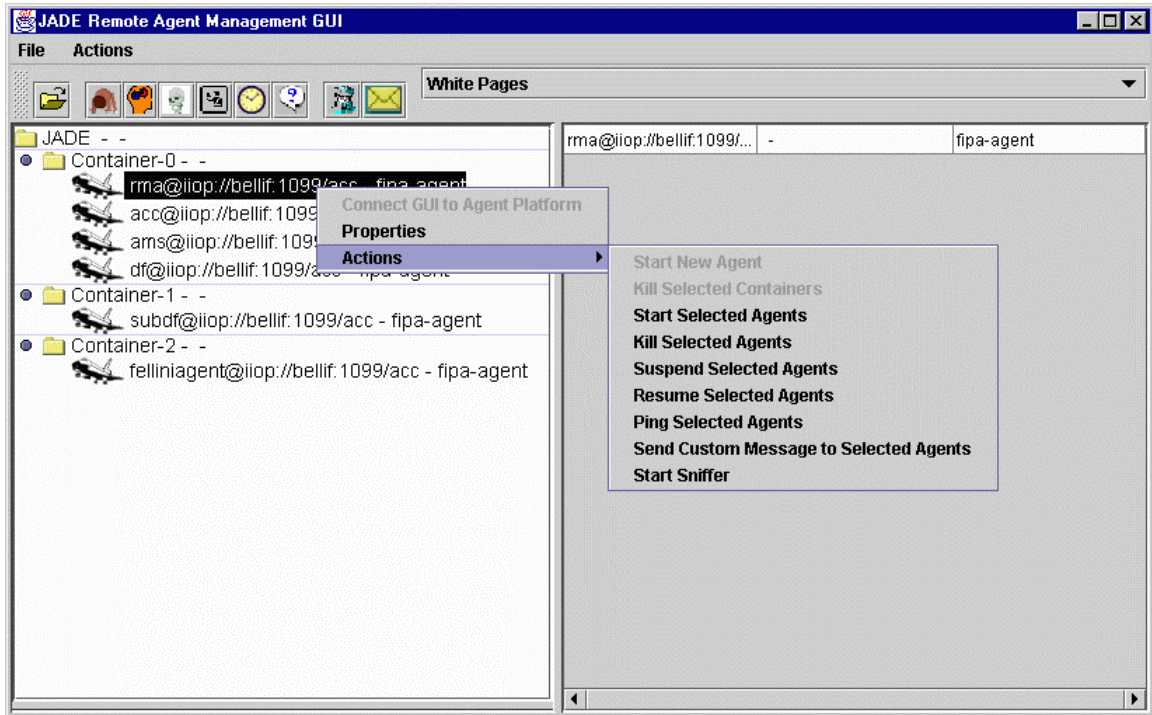

Figure 4 - View of the Agent Platform

3.2.   Communication sub-system

JADE front-end container maintains internally an RMI registry, used by other agent containers at bootstrap time to register themselves with the front-end, thereby joining the Agent Platform. The front-end container maintains a table of all containers along with their RMI object reference; besides, an Agent Global Descriptor Table is kept, which relates each agent name with its AMS data and with its container's RMI object reference.

When a new front-end begins to execute, it creates an internal RMI registry on the current host listening to a user specified TCP/IP port; then it starts FIPA system agents (ACC, AMS and DF).

When a new container begins to execute, it looks up the RMI registry embedded within the running front end of the platform it wants to join; then it retrieves from it the RMI object reference of platform front end. The new agent container registers itself with the front-end and is added to Agent Container Table; then notifies its front-end whenever an agent is created or terminates, in order to keep Agent Global Descriptor Table consistent.

In order to increase performance, each container caches the object reference of other containers as soon as a message is sent to them. This avoids looking-up the Agent Global Descriptor Table every time a message must be sent. However, in order to deal with dynamic situations, where agent containers appear and disappear, if a remote method invocation raises an exception (i.e. either there is cache miss or a cached object reference is stale) then the Agent Global Descriptor Table is looked up again.

When a JADE agent sends a message, the following different cases are possible:

-   If the receiver agent lives on the same agent container, the Java object representing the ACL message is passed to the receiver using an event object, without any message translation (for example, the message sent by Agent1 to Agent2 in Figure 5).

- If the receiver agent lives on the same JADE platform but within a different container, the ACL message is sent using Java Remote Method Invocation framework for distributed object computing. Java RMI allows transparent object marshalling and unmarshalling, avoiding tedious message conversions. Apart from performance, the agent receives a Java object, just like intra-container messaging (for example, the message sent by Agent3 to Agent2 in Figure 5. In this case, a cache hit is supposed to occur, so platform front-end is not contacted).

- If the receiver lives on a different agent platform, standard IIOP protocol and OMG IDL interface are used, according to FIPA standard. This involves translating ACL message object into a character string and then performing a remote invocation using IIOP as middleware protocol. On receiver side, an IIOP unmarshalling will occur, yielding a Java *String* object, which will be parsed into an *ACLMessage* object. Eventually, the Java object will be dispatched to the receiver agent (via Java events or RMI calls).
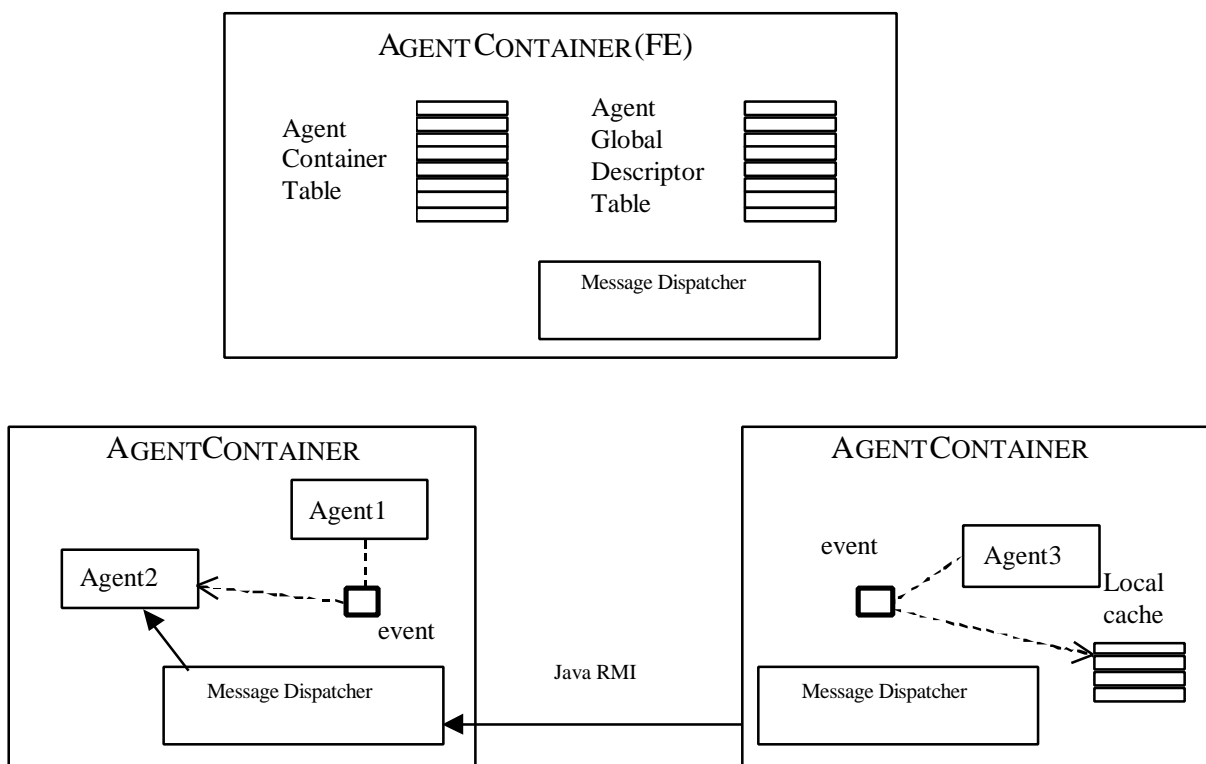
Figure 5 - JADE intra-platform communication model

JADE runtime maintains suitable agent tables and is thus able to choose the most efficient messaging mechanism, according to receiver agent location. Besides, address caching is used on each container to avoid looking up platform front end Global Agent Descriptor Table all the way.

The platform presents a single interface to the outside world using standard *ACC* agent; this agent is a CORBA IIOP server object and listens for remote invocations. Every time it receives an ACL message encoded as a string (usually by non-JADE agents), it parses the message and converts it into a Java *ACLMessage* object used by all JADE agents. It can also perform the dual conversion when a JADE agent sends a message to a non-JADE agent. The *Agent Container* allows JADE to conform to our main assumption that the communication mechanisms must be unknown to all the agents. In fact, every agent thread is a completely independent class, which does not know the details of communication mechanisms and agent platform architecture.

JADE multiple communication model struggles to achieve low cost for message passing; the overheads depend on receiver's location and cache status:

1) **Same container:** No remote invocations; *clone()* is called on ACL message object.

2) **Same Agent Platform, different container, cache hit:** A single RMI call; ACL message object is serialized and unserialized by RMI runtime.

3) **Same Agent Platform, different container, cache miss:** Two RMI calls, the first one to update the cache from Global Agent Descriptor Table and the second one to send the message; an Agent Descriptor object (returned by first call) is serialized and unserialized, and then ACL message object follows the same path.

4) **Different platform (JADE):** A direct call is performed towards remote ACC; this costs a CORBA remote invocation, a double marshalling from Java object to Java *String* and from Java *String* to IIOP byte stream on sender side and a double unmarshalling on receiver side (from IIOP to Java *String* and from Java *String* to Java ACL message object). Then a variable cost is paid in delivering the ACL message from ACC to actual receiver agent; this can be the cost of a case from 1) to 3).

5) **Different platform (non-JADE):** Same as previous case up to IIOP remote invocation. Of course, what actually happens at the other side of the link depends on the other platform implementation.

3.3.   Agent execution model

A distinguishing property of a software agent is its *autonomy*: an agent is not limited to react to external stimuli, but is also able to start new communicative acts autonomously. This requires each agent to have an internal thread of control; however, an agent can engage multiple simultaneous conversations, besides carrying on other activities not involving message exchanges. JADE uses the **Behaviour** abstraction to model the tasks that an agent is able to perform and agents instantiate their behaviours according to the needs and capabilities. From a concurrent programming point of view an agent is an active object, holding inside a thread of control. JADE uses a *thread-per-agent* concurrency model instead of a *thread-per-behaviour* model in order to keep small the number of threads required to run the agent platform. This means that, while different agents run in a preemptive multithreaded environment, two behaviours of the same agent are scheduled cooperatively. Apart from preemption, behaviours work just like co-operative threads, but there is no stack to be saved. A scheduler, implemented by the base Agent class and hidden to the programmer, carries out a round-robin non-preemptive policy among all behaviours available in the ready queue, allowing the execution of a Behaviour-derived class until it will release the execution control by itself. If the task relinquishing the control has not yet completed, it will be rescheduled the next round unless it is blocked; in fact, a behaviour can block itself, for instance while waiting for messages to avoid wasting CPU time and doing busy waiting.

Therefore, the agent developer should extend the Agent class and implement the agent-specific tasks through one or more Behaviour classes, finally instantiate and add them to the agent. The Agent class represents a common superclass for user defined agents. Therefore, from the point of view of the programmer, a JADE agent is simply a Java class that extends the base Agent class. It allows to inherit a basic hidden behaviour (that deals with all agent platform tasks, such as registration, configuration, remote management, …), and a basic set of methods that can be called to implement the application tasks of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, …). Moreover, two more methods are inherited to manage the queue of agent behaviours: addBehaviour(Behaviour) and removeBehaviour(Behaviour).

Behaviour

+ void action() {abstract}
+boolean done(){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 execute() is seen as an atomic computation unit by the scheduler.

SimpleBehaviour

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 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.
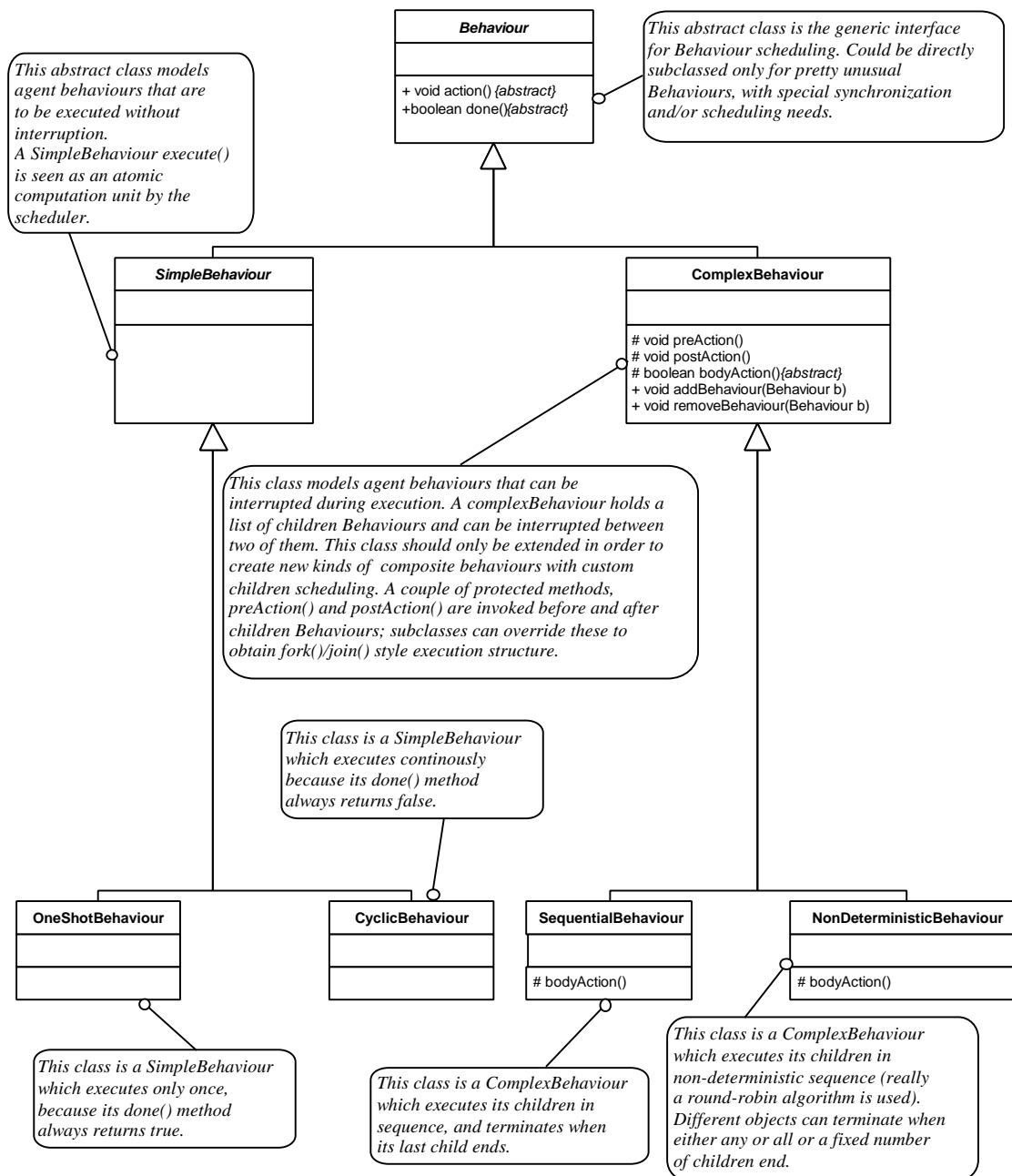
Figure 6 - UML Model of the Behaviour class hierarchy

JADE includes also some ready to use behaviours for the most common tasks in agent programming, such as sending and receiving messages and structuring complex tasks as aggregations of simpler ones (figure 6 is an annotated UML class diagram for JADE behaviours.). Among the others, JADE offers also a so-called *JessBehaviour* that allows full integration with JESS [4], where JADE provides the shell of the agent and guarantees (where possible) the FIPA compliance, while JESS is the engine of the agent that performs all the necessary reasoning.

Behaviour is an abstract class that provides the skeleton of the elementary task to be performed. It exposes two methods: 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 must be used by the agent developer to implement atomic actions of the agent work.

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

OneShotBehaviour is an abstract class that models atomic behaviours that must be executed only once.

CyclicBehaviour is an abstract class that models atomic behaviours that never end and must be executed forever.

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

NonDeterministicBehaviour is a ComplexBehaviour that executes its subBehaviours non deterministically, blocks when all its children are blocked and terminates when a certain condition on its subBehaviours 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 sending a message.

ReceiverBehaviour extends Behaviour and allows receiving a message, which can even be matched against a pattern; the behavior blocks itself (without stopping all other agent activities) if no suitable messages are present in the queue.

4.   CONCLUSIONS

One of the first agent frameworks that consider compliance with the FIPA specifications has been here presented.

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).

Starting from the FIPA assumption that only the external behavior of system components should be specified, while leaving the implementation details and internal architectures to agent developers, a very general agent model has been implemented that can be easily specialized to realize both reactive and BDI architectures. Moreover, the behavior abstraction of our agent model allows simple integration of external software into one of the agent tasks. The implementation of the JessBehaviour allows the usage of JESS as the agent-reasoning engine. An implementation practice that we have found useful is the usage of JESS to control the activation and de-activation of the JADE Behaviours by implementing, as a consequence, a mixed reactive-deliberative agent architecture (where JESS plays the deliberative role and the JADE behaviours play the reactive role).

JADE is a trademark registered by CSELT and it is mainly the result of a research activity. Some Java packages have been implemented within the framework of the ACTS AC317 "FACTS" project. Version 0.93 of JADE has been distributed to some partners of this project (that we thank for their patience and feedback) for testing and evaluation. In particular, two applications are being prototyped in the project by using JADE. Both share the ultimate application goal of resource bundling, where one prototype analyses the television entertainment domain and the other one analyses the travel application domain. In both cases, agents are used to represent information resources, service providers and provider's interests. A middle layer of brokerage, implemented by agents, hides to the users differences between providers and helps them in locating the best ones. Finally, agents are also used to represent user preferences and to act pro-actively, on behalf of the user, to search, reserve, and possibly buy, information and services. The feedback received so far in using JADE is rather positive: developers have found very useful the availability of such a form of agent middle-ware and development framework; it allowed them to actually concentrate on the realization of their application-specific tasks rather than on the agent management. Some bugs were discovered, reported and fixed so that the overall reliability and usability of JADE was improved.

JADE tries to provide its users with standard agent technologies while keeping runtime overheads low. Particularly, a "pay as you go" design philosophy was applied, i.e. user defined agents incur in runtime costs associated with a JADE feature only when they actually use that feature.

For example, multiple communication means are used for ACL message transport, selecting the one among them with the lowest associated cost. Besides, cooperative behaviour scheduling was preferred over true intra-agent multithreading because of reduced synchronization and scheduling costs: since an agent's knowledge is shared among all its behaviours, preemptive behaviour scheduling would require Java synchronized methods to be used all the way, thus paying a significant performance penalty (Java synchronized methods are about 100 times slower than ordinary methods, due to object lock management overhead [2]).

JADE tries to be efficient also with respect to resource consumption: again, cooperative behaviours help reduce overall thread number. Besides, many JADE objects can be recycled instead of destroyed and recreated back, reducing dynamic memory allocation (a 'new' call costs about 150 method invocations [2])

Of course, proactive design decisions are not enough to grant a really efficient system. Thorough performance measurements and evaluation is required. We could not carry out any of these simply because JADE is still a developing product, but plan to profile and further optimize our framework soon. However, it is to say that ever since version 0.6 JADE had users outside its development team: thanks to their interest and feedback some performance and correctness problems were early reported and fixed. This allows ourselves to feel confident enough about JADE capabilities and performance, while still continuing to improve it.

The development of JADE is still continuing. Further improvements, enhancements, and implementations have already been planned, included support for agent mobility as specified by FIPA98. At January meeting of FIPA, in Seoul, JADE is going to participate to the platform interoperability tests organized by FIPA. Four levels of interoperability are going to be tested between four different implementations of the FIPA specifications: JADE, MECCA [7] from Siemens, ASL [1] from Broadcom, and the COMTEC [12] agent platform:

- at the lower level, agent naming and message transport between different agent platforms will be tested (*platform-to-platform*);
- then, agent registration with foreign platform, and in particular, with foreign DF will be tested (*agent-to-platform*);

- then, it will be tested message exchanging between agents belonging to different platforms (*agent-to-agent*);

- finally, the maximum level of interoperability will also be tested with a simple application, appointments scheduling, where agents from different platforms and implementations will cooperate together by using high-level interaction protocols, like contract-net, in order to schedule meetings (*application-to-application)*.

5.   REFERENCES

[1]   Broadcom Eireann Research Ltd. ASL – Agent Service Layer. 1998. Available from http://www.broadcom.ie/asl.

[2]   B. Eckel. Thinking in Java. , Upper Sandle River, NJ. Prentice Hall, 1998.

[3]   T. Finin and Y. Labrou. KQML as an agent communication language. . J.M. In: Bradshaw (ed.), Software Agents, pp. 291-316. Cambridge, MA, 1997.

[4]   E.J. Firedman-Hill. Java Expert System S hell. 1998. Available from http://herzberg.ca.sandia.gov/jess

[5]   Foundation for Intelligent Physical Agents. Specifications. 1997. Available from *http://www.fipa.org*

[6]   M.R. Genesereth and S.P. Ketchpel. Software Agents. Comm. of ACM, 37(7):48-53.1994.

[7]   A.D. Lux and D. Steiner. Understanding Cooperation: an Agent's Perspective, in Proc. ICMAS'95. San Francisco, USA. 1995.

[8]   H.S. Nwana, D.T. Ndumu and L.C. Lee. ZEUS: An advanced Tool-Kit for Engineering Distributed Mulyi-Agent Systems. In: Proc. of PAAM98, pp. 377-391, London, U.K. 1998.

[9]   Object Management Group. 95-11-03: Common Facilities RFP3 Final Draft. 1995. Available from http://www.omg.org/docs/1995/95-11-03.ps.

[10] R.S. Patil, R.E. Fikes, P.F. Patel-Scheneider, D. McKay, T. Finin, T. Gru ber and R. Neches. The DARPA knowledge sharing effort: progress report. In: Proc. Third Conf. on Principles of Knowledge Representation and Reasoning, pp 103-114. Cambridge, MA. 1992.

[11] M. Straßer, J. Baumann and F. Hohl (1997): Mole - A Java based Mo bile Agent System. In: M. Mühlhäuser: (ed.), Special Issues in Object Oriented Programming. dpunkt Verlag, pp. 301-308, 1997.

[12] H.   Suguri.   COMTEC   Agent   Platform.   1998.   Available   from http://www.fipa.org/glointe.htm

[13] K. Sycara, A. Pannu, M. Willia mson and D. Zeng. Distributed Intelligent Agents. IEEE Expert, 11(6): 36-46. 1996.