Developing Multi-agent Systems with JADE

Fabio Bellifemine[1], Agostino Poggi[2], and Giovanni Rimassa[2]

[1] CSELT S.p.A.
Via G. Reiss Romoli, 274, 10148, Torino, Italy
bellifemine@cselt.it

[2] Dipartimento di Ingegneria dell'Informazione, University of Parma
Parco Area delle Scienze, 181A, 43100, Parma, Italy
(poggi,rimassa}@ce.unipr.it

Abstract. JADE (Java Agent Development Framework) is a software framework to make easy the development of multi-agent applications in compliance with the FIPA specifications. JADE can then be considered a middle-ware that implements an efficient agent platform and supports the development of multi agent systems. JADE agent platform tries to keep high the performance of a distributed agent system implemented with the Java language. In particular, its communication architecture tries to offer flexible and efficient messaging, transparently choosing the best transport available and leveraging state-of-the-art distributed object technology embedded within Java runtime environment. JADE uses an agent model and Java implementation that allow good runtime efficiency, software reuse, agent mobility and the realization of different agent architectures.

1. Introduction

Nowadays, agent-based technologies are considered the most promising means to deploy enterprise-wide and worldwide applications that often must operate across corporations and continents and inter-operate with other heterogeneous systems. It is because they offer the high-level software abstractions needed to manage complex applications and because they were invented to cope with distribution and interoperability [2,9,12,19,24,36].

However, agent-based technologies are still immature and few truly agent-based systems have been built. Agent-based technologies cannot keep their promises, and will not become widespread, until standards to support agent interoperability are in place and adequate environments for the development of agent systems are available. However, many people are working on the standardisation of agent technologies (see, for example, the work done by Knowledge Sharing Effort [27], OMG [22] and FIPA [7]) and on development environments to build agent systems (see, for example, DMARS [28], RETSINA [34] MOLE [1]).

Such environments provide some predefined agent models and tools to ease the development of systems. Moreover, some of them try to inter-operate with other agent systems through a well-known agent communication language, that is, KQML [6]. However, a shared communication language is not enough to support interoperability

C. Castelfranchi, Y. Lespérance (Eds.): Intelligent Agents VII, LNAI 1986, pp. 89–103, 2001.

between different agent systems, because common agent services and ontology are also needed. The standardisation work of FIPA acknowledges this issue and, beyond an agent communication language, specifies the key agents necessary for the management of an agent system and the shared ontology to be used for the interaction between two systems.

In this paper, we present JADE (Java Agent Development Framework), a software framework to write agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. The next section introduces FIPA specifications. Section three introduces related work on software frameworks to develop agent systems. Section four describes JADE main features. Section five describes the architecture of the agent platform, the communication subsystem. Section six presents JADE agent model. Finally, section seven concludes with a brief description about JADE main features, the use of JADE to realise applications and the relationships between JADE and some other agent software frameworks.

2. FIPA Specifications

The Foundation for Intelligent Physical Agents (FIPA) [7] is an international non-profit association of companies and organisations sharing the effort to produce specifications for generic agent technologies. FIPA does not just promote a technology for a single application domain but a set of general technologies for different application areas that developers can integrate to make complex systems with a high degree of interoperability.

The first output documents of FIPA, named FIPA97 specifications, state the normative rules that allow a society of agents to exist, operate and be managed. First of all they describe the reference model of an agent platform: they identify the roles of some key agents necessary for managing the platform, and describe the agent management content language and ontology. Three 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 maintaining a directory of resident agents and for handling their life cycle. The Agent Communication Channel (ACC) provides the path for basic contact between agents inside and outside the platform. The ACC is the default communication method, which offers a reliable, orderly and accurate message routing service. FIPA97 mandates ACC support for IIOP in order to inter-operate with other compliant agent platforms. The Directory Facilitator (DF) is the agent that provides yellow page services to the agent platform.

The specifications also define the Agent Communication Language (ACL), used by agents to exchange messages. FIPA ACL is a language describing message encoding and semantics, but it does not mandate specific mechanisms for message transportation. Since different agents might run on different platforms on different networks, messages are encoded in a textual form, assuming that agents are able to transmit 7-bit data. ACL syntax is close to the widely used communication language KQML. However, there are fundamental differences between KQML and ACL, the most evident being the existence of a formal semantics for FIPA ACL, which should eliminate any ambiguity and confusion from the usage of the language.

FIPA supports common forms of inter-agent conversations through *interaction protocols*, which are communication patterns followed by two or more agents. Such protocols range from simple query and request protocols, to more complex ones, as the well-known contract net negotiation protocol and English and Dutch auctions.

The remaining parts of the FIPA specifications deal with other aspects, in particular with agent-software integration, agent mobility, agent security, ontology service, and human-agent communication. However they are not described here because they have not yet been considered in the JADE implementation. The interested reader should refer directly to the FIPA Web page [7].

3. Related Work

A lot of research and commercial organisations are involved in the realisation of agent applications and a considerable number of agent construction tools has been realised [29]. Some of the most interesting are AgentBuilder [30], AgentTool [4], ASL [16], Bee-gent [15], FIPA-OS [23], Grasshopper-2 [10], MOLE [1], the Open Agent Architecture [20], RETSINA [34] and Zeus [25].

AgentBuilder [30] is a tool for building Java agent systems based on two components: the Toolkit and the Run-Time System. The Toolkit includes tools for managing the agent software development process, while the Run-Time System provides an agent engine, that is, an interpreter, used as execution environment of agent software. AgentBuilder agents are based on a model derived by the Agent-0 [32] and PLACA [35] agent models.

AgentTool [4] is a graphical environment to build heterogeneous multi-agent systems. It is a kind of CASE tool, specifically oriented towards agent-oriented software engineering, whose major advantages are the complete support for the MaSE methodology (developed by the same authors together with the tool) and the independence from agent internal architecture (with MaSE and agentTool it is possible to build multi agent systems made of agents with different internal architectures).

ASL [16] is an agent platform that supports the development in C/C++, Java, JESS, CLIPS and Prolog. ASL is built upon the OMG's CORBA 2.0 specifications. The use of CORBA technology facilitates seamless agent distribution and allows adding to the platform the language bindings supported by the used CORBA implementations. Initially, ASL agents used to communicate through KQML messages, now the platform is FIPA compliant supporting FIPA ACL.

Bee-gent [15] is a software framework to develop agent systems compliant to FIPA specification that has been realised by Toshiba. Such a framework provides two types of agents: wrapper agents used to agentify existing applications and mediation agents supporting the wrappers coordination by handling all their communications. Bee-gent also offers a graphic RAD tool to describe agents through state transition diagrams and a directory facility to locate agents, databases and applications.

FIPA-OS [23] is another software framework to develop agent systems compliant to FIPA specification that has been realised by NORTEL. Such a framework provides the mandatory components realising the agent platform of the FIPA reference model (i.e., the AMS, ACC and DF agents, and an internal platform message transport

system), an agent shell and a template to produce agents that communicate taking advantage of FIPA-OS agent platform.

Grasshopper-2 [10] is a pure Java based Mobile Agent platform, conformant to existing agent standards, as defined by the OMG - MASIF (Mobile Agent System Interoperability Facility) [22] and FIPA specifications. Thus Grasshopper-2 is an open platform, enabling maximum interoperability and easy integration with other mobile and intelligent agent systems. The Grasshopper-2 environment consists of several Agencies and a Region Registry, remotely connected via a selectable communication protocol. Several interfaces are specified to enable remote interactions between the distinguished distributed components. Moreover, Grasshopper-2 provides a Graphical User for user-friendly access to all the functionality of an agent system.

MOLE [1] is an agent system developed in Java whose agents do not have a sufficient set of features to be considered truly agent systems [9,33]. However, MOLE is important because it offers one of the best supports for agent mobility. Mole agents are multi-thread entities identified by a globally unique agent identifier. Agents interact through two types of communication: through RMI for client/server interactions and through message exchanges for peer-to-peer interactions.

The Open Agent Architecture [20] is a truly open architecture to realise distributed agent systems in a number of languages, namely C, Java, Prolog, Lisp, Visual Basic and Delphi. Its main feature is its powerful facilitator that coordinates all the other agents in their tasks. The facilitator can receive tasks from agents, decompose them and award them to other agents.

RETSINA [34] offers reusable agents to realise applications. Each agent has four modules for communicating, planning, scheduling and monitoring the execution of tasks and requests from other agents. RETSINA agents communicate through KQML messages.

Zeus [25] allows the rapid development of Java agent systems by providing a library of agent components, by supporting a visual environment for capturing user specifications, an agent building environment that includes an automatic agent code generator and a collection of classes that form the building blocks of individual agents. Agents are composed of five layers: API layer, definition layer, organisational layer, coordination layer and communication layer. The API layer allows the interaction with non-agentized world.

4. JADE

JADE (Java Agent Development Environment) is a software framework to make easy the development of agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. JADE is an Open Source project, and the complete system can be downloaded from JADE Home Page [11]. 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 default 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. 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 lightweight communication between agents on the same host. Parallel tasks can be still executed by one agent, and JADE schedules these tasks in a cooperative way.
- A number of FIPA-compliant additional DFs (Directory Facilitator) can be started at run time in order to build multi-domain environments, where a domain is a logical set of agents, whose services are advertised through a common facilitator.
- Java API to send/receive messages to/from other agents; ACL messages are represented as ordinary Java objects.
- FIPA97-compliant IIOP protocol to connect different agent platforms.
- Lightweight 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.
- Library of FIPA interaction protocols ready to be used.
- Support for agent mobility within a JADE agent platform.
- Library to manage user-defined ontologies and content languages.
- Graphical user interface to manage several agents and agent platforms from the same agent. The activity of each platform can be monitored and logged. All life cycle operations on agents (creating a new agent, suspending or terminating an existing agent, etc.) can be performed through this administrative GUI.

The JADE system can be described from two different points of view. On the one hand, JADE is a runtime system for FIPA-compliant Multi Agent Systems, supporting application agents whenever they need to exploit some feature covered by the FIPA standard specification (message passing, agent life-cycle management, etc.). On the other hand, JADE is a Java framework for developing FIPA-compliant agent applications, making FIPA standard assets available to the programmer through object oriented abstractions. The two following subsections will present JADE from the two standpoints, trying to highlight the major design choices followed by the JADE development team. A final discussion section will comment on JADE actual strengths and weaknesses and will describe the future improvements envisaged in the JADE development roadmap.

5. JADE Runtime System

A running agent platform must provide several services to the applications: when looking at the parts 1 and 2 of the FIPA97 specification, is can be seen that these services fall into two main areas, that is, message passing support with FIPA ACL and agent management with life-cycle, white and yellow pages, etc.

5.1 Distributed Agent Platform

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

While appearing as a single entity to the outside world, a JADE agent platform is itself a distributed system, since it can be split over several hosts with one among them acting as a front end for inter-platform IIOP communication. A JADE system is made by one or more *Agent Container*, each one living in a separate Java Virtual Machine and communicating using Java RMI. IIOP is used to forward outgoing messages to foreign agent platforms. A special, *Front End* container is also an IIOP server, listening at the official agent platform ACC address for incoming messages from other platforms. Figure 1 shows the architecture of a JADE Agent Platform.

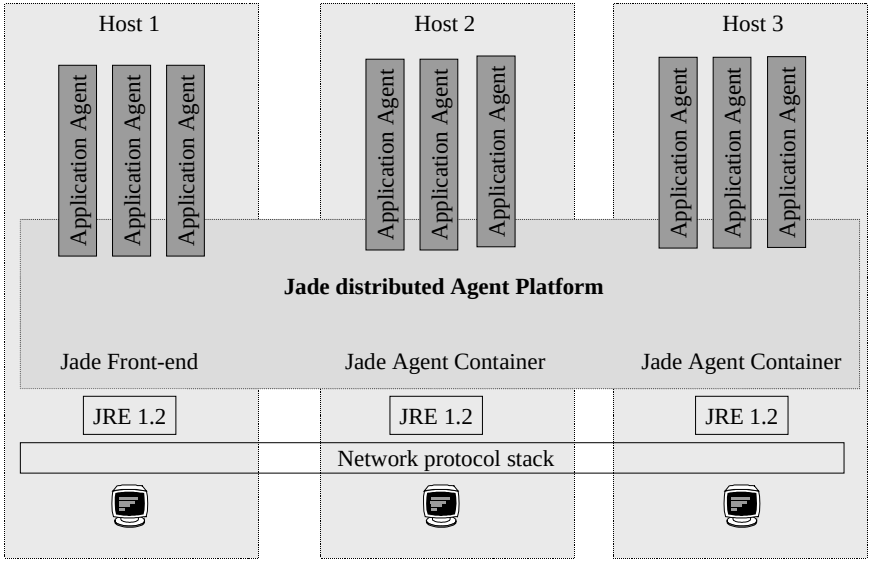

Fig. 1. Software architecture of a JADE Agent Platform

5.2 Message Delivery Subsystem

FIPA agent communication model is peer-to-peer though multi-message context is provided by interaction protocols and conversation identifiers. On the other hand, JADE uses transport technologies such as RMI, CORBA and event dispatching which are typically associated with reactive systems. Clearly, there is some gap to bridge to map the explicitly addressed FIPA message-passing model into the request/response communication model of distributed objects. This is why in JADE ordinary agents are not distributed objects, but agent containers are.

A software agent, in compliance to FIPA agent model, has a *globally-unique identifier* (*GUID*), that can be used by every other agent to address it with ACL messages; likewise, an agent will put its GUID into the *:sender* slot of ACL messages it sends around. So, JADE must figure out receiver location by simply looking at *:receiver* message slot. Since a FIPA97 GUID resembles an email address, it has the form: <*agent name*> @ <*platform address*>, it is fairly easy to recover the agent name and the platform address from it.

When an ACL message is sent to a software agent, three options are given:

- *Receiver on the same container of the same platform:* Java events are used, the *ACLMessage* is simply cloned.
- *Receiver on a different container of the same platform:* Java RMI is used, the message is serialised at sender side, a remote method is called and the message is unserialised at receiver side.
- *Receiver on a different platform:* IIOP is used, the *ACLMessage* is converted into a *String* and marshalled at sender side, a remote CORBA call is done and an unmarshalling followed by ACL parsing occurs at receiver side.

5.3 Address Management and Caching

JADE tries to select the most convenient of the three transport mechanisms above according to agents location. Basically, each container has a table of its local agents, called the *Local-Agent Descriptor Table* (*LADT*), whereas the front-end, besides its own LADT, also maintains a *Global-Agent Descriptor Table* (*GADT*), mapping every agent into the RMI object reference of its container. Moreover, JADE uses an address caching technique to avoid querying the front-end continuously for address information.

Besides being efficient, this is also meant to support agent mobility, where agent addresses can change over time (e.g. from local to RMI); transparent caching means that messaging subsystem will not be affected when agent mobility will be introduced into JADE. Moreover, if new remote protocols will be needed in JADE (e.g. a wireless protocol for nomadic applications), they will be seamlessly integrated inside the messaging and address caching mechanisms.

5.4 Mobility

The new JADE version adds the support for agent mobility. Exploiting Java Serialization API and dynamic class loading, it is now possible to move or clone a JADE agent over different containers but within the same JADE agent platform. Our current implementation is completely proprietary and does not allow inter-platform mobility over the FIPA IIOP standard message transport service. While a more complete mobility support could be possible, we feel that it would not be worth the effort, because FIPA specifications for mobility support is still incomplete and a proprietary, JADE-only mobility service would not help standardization and interoperability.

Rather, some more general proposals should be submitted to FIPA, undergoing public discussion and evaluation. Then, an effective and interoperable implementation could be built.

5.5 User-Defined Ontologies and Content Languages

According to the FIPA standard, achieving agent level interoperability requires that different agents share much more than a simple on-the-wire protocol. While FIPA mandates a single agent communication language, the *FIPA ACL*, it explicitly allows application dependent content languages and ontologies. The FIPA specifications themselves now contain a *Content Language Library*, whereas various mandatory ontologies are defined and used within the different parts of the FIPA standard.

The last version of JADE lets application programmers create their own content languages and their ontologies. Every JADE agent keeps a capability table where the known languages and ontologies are listed; user defined codecs must be able to translate back and forth between the *String* format (according to the content language syntax) and a frame based representation.

If a user-defined ontology is defined, the application can register a suitable Java class to play an ontological role, and JADE is able to convert to and from frames and user defined Java objects. Acting this way, application programmers can represent their domain specific concepts as familiar Java classes, while still being able to process them at the agent level (put them within ACL messages, reasoning about them, etc.).

5.6 Tools for Platform Management and Monitoring

Beyond a runtime library, JADE offers some tools to manage the running agent platform and to monitor and debug agent societies; all these tools are implemented as FIPA agents themselves, and they require no special support to perform their tasks, but just rely on JADE AMS.

The general management console for a JADE agent platform is called *RMA* (*Remote Monitoring Agent*). The RMA acquires the information about the platform and executes the GUI commands to modify the status of the platform (creating agents, shutting down containers, etc.) through the AMS. The Directory Facilitator agent also has a GUI, with which it can be administered, configuring its advertised agents and services.

JADE users can debug their agents with the *Dummy Agent* and the *Sniffer Agent*.

The Dummy Agent is a simple tool for inspecting message exchanges among agents, facilitating validation of agent message exchange patterns and interactive testing of an agent.

The Sniffer Agent allows to track messages exchanged in a JADE agent platform: every message directed to or coming from a chosen agent or group is tracked and displayed in the sniffer window, using a notation similar to UML *Sequence Diagrams*.

6. JADE Agent Development Model

FIPA specifications state nothing about agent internals, but when JADE was designed and built they had to be addressed. A major design issue is the execution model for an agent platform, both affecting performance and imposing specific programming styles on agent developers. As will be shown in the following, JADE solution stems from the balancing of forces from ordinary software engineering guidelines and theoretical agent properties.

6.1 From Agent Theory to Class Design

A distinguishing property of a software agent is its *autonomy*; an agent is not limited to react to external stimuli, but it's also able to start new communicative acts of its own. A software agent, besides being autonomous, is said to be *social*, because it can interact with other agents in order to pursue its goals or can even develop an overall strategy together with its peers.

FIPA standard bases its *Agent Communication Language* on *speech-act theory* [31] and uses a mentalistic model to build a formal semantic for the *performatives* agents exchange. This approach is quite different from the one followed by distributed objects and rooted in *Design by Contract* [21]; a fundamental difference is that invocations can either succeed or fail but a *request* speech act can be refused if the receiver is unwilling to perform the requested action.

Trying to map the aforementioned agent properties into design decisions, the following list was produced:

- *Agents are autonomous*, then they are active objects.
- *Agents are social*, then intra-agent concurrency is needed.
- *Messages are speech acts*, then asynchronous messaging must be used.
- *Agents can say "no"*, then peer-to-peer communication model is needed.

The autonomy property requires each agent to be an *active object* [17] with at least a Java thread, to proactively start new conversations, make plans and pursue goals. The need for sociality has the outcome of allowing an agent to engage in many conversations simultaneously, dealing with a significant amount of concurrency.

The third requirement suggests asynchronous message passing as a way to exchange information between two independent agents, that also has the benefit of producing more reusable interactions [33]. Similarly, the last requirement stresses that in a Multi Agent System the sender and the receiver are equals (as opposed to client/server systems where the receiver is supposed to obey the sender). An autonomous agent should also be allowed to ignore a received message as long as he wishes; this advocates using a *pull consumer* messaging model [26], where incoming messages are buffered until their receiver decides to read them.

6.2 JADE Agent Concurrency Model

The above considerations help in deciding how many threads of control are needed in an agent implementation; the autonomy requirement forces each agent to have at least a thread, and the sociality requirement pushes towards many threads per agent. Unfortunately, current operating systems limit the maximum number of threads that

can be run effectively on a system. JADE execution model tries to limit the number of threads and has its roots in actor languages.

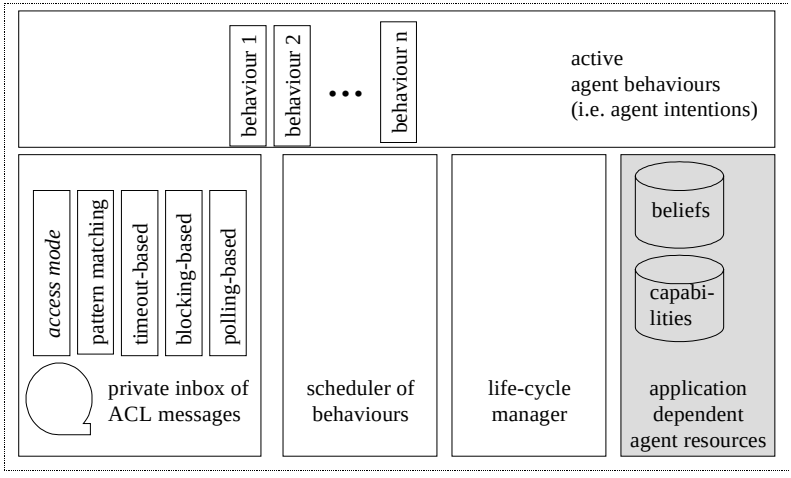

Fig. 2. JADE agent architecture.

The *Behaviour* abstraction models agent tasks: a collection of behaviours are scheduled and executed to carry on agent duties (see figure 2). Behaviours represent logical threads of a software agent implementation. According to *Active Object* design pattern [17], every JADE agent runs in its own Java thread, satisfying autonomy property; instead, to limit the threads required to run an agent platform, all agent behaviours are executed cooperatively within a single Java thread. So, JADE uses a *thread-per-agent* execution model with cooperative intra-agent scheduling.

JADE agents schedule their behaviour with a "*cooperative scheduling on top of the stack*", in which all behaviours are run from a single stack frame (*on top of the stack*) and a behaviour runs until it returns from its main function and cannot be pre-empted by other behaviours (*cooperative scheduling*).

JADE model is an effort to provide fine-grained parallelism on coarser grained hardware. A likewise, stack based execution model is followed by Illinois Concert runtime system [14] for parallel object oriented languages. Concert executes concurrent method calls optimistically on the stack, reverting to real thread spawning only when the method is about to block, saving the context for the current call only when forced to.

Choosing not to save behaviour execution context means that agent behaviours start from the beginning every time they are scheduled for execution. So, behaviour state that must be retained across multiple executions must be stored into behaviour instance variables. A general rule for transforming an ordinary Java method into a JADE behaviour is:

1. *Turn the method body into an object whose class inherits from Behaviour.*
2. *Turn method local variables into behaviour instance variables.*
3. *Add the behaviour object to agent behaviour list during agent start-up.*

The above guidelines apply the *reification technique* [13] to agent methods, according to *Command* design pattern [18]; an agent behaviour object reifies both a method and a separate thread executing it. A new class must be written and instantiated for every agent behaviour, and this can lead to programs harder to understand and maintain. JADE application programmers can compensate for this shortcoming using Java *Anonymous Inner Classes*; this language feature makes the code necessary for defining an agent behaviour only slightly higher than for writing a single Java method.

JADE *thread-per-agent* model can deal alone with the most common situations involving only agents: this is because every JADE agent owns a single message queue from which ACL messages are retrieved. Having multiple threads but a single mailbox would bring no benefit in message dispatching. On the other hand, when writing agent wrappers for non-agent software, there can be many interesting events from the environment beyond ACL message arrivals. Therefore, application developers are free to choose whatever concurrency model they feel is needed for their particular wrapper agent; ordinary Java threading is still possible from within an agent behaviour.

6.3 Using Behaviours to Build Complex Agents

The developer implementing an agent must extend *Agent* class and implement agent-specific tasks by writing one or more *Behaviour* subclasses. User defined agents inherit from their superclass the capability of registering and deregistering with their platform and a basic set of methods (e.g. send and receive ACL messages, use standard interaction protocols, register with several domains). Moreover, user agents inherit from their *Agent* superclass two methods: *addBehaviour(Behaviour)* and *removeBehaviour(Behaviour)*, to manage the behaviour list of the agent.

JADE contains ready made behaviours for the most common tasks in agent programming, such as sending and receiving messages and structuring complex tasks as aggregations of simpler ones. For example, JADE offers a so-called *JessBehaviour* that allows full integration with JESS [8], a scripting environment for rule programming offering an engine using the Rete algorithm to process rules.

Behaviour is an abstract class that provides the skeleton of the elementary task to be performed. It exposes three methods: the *action()* method, representing the "true" task to be accomplished by the specific behaviour classes; the *done()* method, used by the agent scheduler, that must return *true* when the behaviour has finished and *false* when the behaviour has not and the *action()* method must be executed again; the *reset()* method, used to restart a behaviour from the beginning.

JADE follows a compositional approach to allow application developers to build their own behaviours out of the simpler ones directly provided by the framework. Applying the *Composite* design pattern, *ComplexBehaviour* class is itself a *Behaviour*, with some sub-behaviours or *children*, defining two methods *addSubBehaviour(Behaviour)* and *removeSubBehaviour(Behaviour)*. This permits agent writers to implement a structured tree with behaviours of different kinds. Besides *ComplexBehaviour*, JADE framework defines some other subclasses of *Behaviour*: *SimpleBehaviour* can be used to implement atomic steps of the agent work. A behaviour implemented by a subclass of *SimpleBehaviour* is executed by JADE scheduler in a single time frame. Two more subclasses to send and receive

messages are *SenderBehaviour* and *ReceiverBehaviour*. They can be instantiated passing appropriate parameters to their constructors. *SenderBehaviour* allows sending a message, while *ReceiverBehaviour* allows receiving a message, which can be matched against a pattern; the behaviour blocks itself (without stopping all other agent activities) if no suitable messages are present.

JADE recursive aggregation of behaviour objects resembles the technique used for graphical user interfaces, where every interface widget can be a leaf of a tree whose intermediate nodes are special container widgets, with rendering and children management features. An important distinction, however, exists: JADE behaviours reify execution tasks, so task scheduling and suspension are to be considered, too.

Thinking in terms of software patterns, if *Composite* is the main structural pattern used for JADE behaviours, on the behavioural side we have *Chain of Responsibility*: agent scheduling directly affects only top-level nodes of the behaviour tree, but every composite behaviour is responsible for its children scheduling within its time frame.

7. Conclusions

JADE design tries to put together abstraction and efficiency, giving programmers easy access to the main FIPA standard assets while incurring into runtime costs for a feature only when that specific feature is used. This "*pay as you go*" approach drives all the main JADE architectural decisions: from the messaging subsystems that transparently chooses the best transport available, to the address management module, that uses optimistic caching and direct connection between containers.

Since JADE is a middleware for developing distributed applications, it must be evaluated with respect to scalability and fault tolerance, which are two very important issues for distributed robust software infrastructures.

When discussing scalability, it is necessary to first state with respect to which variable; in a Multi Agent System, the three most interesting variables are the number of agents in a platform, the number of messages for a single agent and the number of simultaneous conversations a single agent gets involved in.

JADE tries to support large Multi Agent Systems as possible; exploiting JADE distributed architecture, clusters of related agents can be deployed on separate agent containers in order to reduce both the number of threads per host and the network load among hosts.

JADE scalability with respect to the number of messages for a single agent is strictly dependent on the lower communication layers, such as the CORBA ORB used for IIOP and the RMI transport system. Again, the distributed platform with decentralised connection management tries to help; when an agent receives many messages, only the ones sent by remote agents stress the underlying communication subsystem, while messages from local agents travel on a fast path of their own.

JADE agents are very scalable with respect to the number of simultaneous conversations a single agent can participate in. This is in fact the whole point of the two level scheduling architecture: when an agent engages in a new conversation, no new threads are spawned and no new connections are set up, just a new behaviour object is created. So the only overhead associated to starting conversations is the behaviour object creation time and its memory occupation; agents particularly

sensitive to these overheads can easily bound them a priori implementing a behaviour pool.

From a fault tolerance standpoint, JADE does not perform very well due to the single point of failure represented by the Front End container and, in particular, by the AMS. A replicated AMS would be necessary to grant complete fault tolerance of the platform. Nevertheless, it should be noted that, due to JADE decentralised messaging architecture, a group of cooperating agents could continue to work even in the presence of an AMS failure. What is really missing in JADE is a restart mechanism for the front-end container and the FIPA system agents.

Even if JADE is a young project, it has been designed with criteria more academics than industrials, and even if only recently it has been released under Open Source License, it has been already used into some of projects.

FACTS [5] is a project in the framework of the ACTS programme of the European Commission that has used JADE in two application domains. In the first application domain, JADE provides the basis for a new generation TV entertainment system. The user accesses a multi-agent system to help him on the basis of his profile that is able to capture, model, and refine over-time through the collaboration of agents with different capabilities. The second application domain deals with agents collaborating, and at the same time competing, in order to help the user to purchase a business trip. A Personal Travel Assistance represents the user interests and cooperates with a Travel Broker Agent in order to select and recommend the business trip.

CoMMA [3] is a project in the framework of the IST programme of the European Commission that is using JADE to help users in the management of an organisation corporate memory and in particular to facilitate the creation, dissemination, transmission and reuse of knowledge in the organisation.

JADE offers an agent model that is more “primitive” than the agents models offered, for example, by AgentBuilder, dMARS, RETSINA and Zeus; however, the overhead due to such sophisticated agent models might not be justified for agents that must perform some simple tasks. Starting from FIPA assumption that only the external behavior of system components should be specified, leaving the implementation details and internal architectures to agent developers, we realize a very general agent model that can be easily specialized to implement, for example, reactive or BDI architectures or other sophisticated architectures taking also advantages of the separation between computation and synchronization code inside agent behaviours through the use of guards and transitions. In particular, we can realize a system composed of agents with different architectures, but able to interact because on the top of the “primitive” JADE agent model. Moreover, the behavior abstraction of our agent model allows an easy integration of external software. For example, we realized a *JessBehaviour* that allows the use of JESS [8] as agent reasoning engine.

Acknowledgements. Thanks to all the people that contributed to development of JADE. The work has been partially supported by a grant from CSELT, Torino.

References

1. J. Baumann, F. Hohl, K. Rothermel and M. Straßer. Mole - Concepts of a Mobile Agent System, World Wide Web,1(3):123-137, 1998.
2. J.M. Bradshaw. Software Agents. MIT Press, Cambridge, MA, 1997.
3. CoMMA Project Home Page. Available at http://www.ii.atos-group.com/sophia/comma/HomePage.htm.
4. S.A. DeLoach and M. Wood, Developing Multiagent Systems with agentTool. In C. Castelfranchi and Y. Lespérance, editors, Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th International Workshop, ATAL-2000, Boston, MA, USA, July 7-9, 2000, Proceedings, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.
5. FACTS Project Home Page: http://www.labs.bt.com/profsoc/facts/.
6. T. Finin and Y. Labrou. KQML as an agent communication language. In: J.M. Bradshaw (ed.), Software Agents, pp. 291-316. MIT Press, Cambridge, MA, 1997.
7. Foundation for Intelligent Physical Agents. Specifications. 1999. Available at http://www.fipa.org.
8. E.J. Friedman-Hill. Java Expert System Shell. 1998. Available At http://herzberg.ca.sandia.gov/jess.
9. M.R. Genesereth and S.P. Ketchpel. Software Agents. Comm. of ACM, 37(7):48-53.1994.
10. Grasshopper Home Page. Available at http://www.ikv.de/products/grasshopper.
11. The JADE Project Home Page, 2000. Available at http://sharon.cselt.it/projects/jade.
12. N.R. Jennings and M. Wooldrige. Agent Technology: Foundations, Applications, and Markets. Stringer, Berlin, Germany, 1998.
13. R.E. Johnson and J.M. Zweig. Delegation in C++. The Journal of Object Oriented Programming, 4(7):31-34, 1991.
14. V. Karamcheti, J. Plevyak and A. Chien. Runtime Mechanisms for Efficient Dynamic Multithreading. Journal of Parallel and Distributed Computing, 37:21-40, 1996.
15. T. Kawamura, N. Yoshioka, T. Hasegawa, A. Ohsuga and S. Honiden. Bee-gent : Bonding and Encapsulation Enhancement Agent Framework for Development of Distributed Systems. Proceedings of the 6th Asia-Pacific Software Engneering Conference, 1999.
16. D. Kerr, D. O'Sullivan, R. Evans, R. Richardson and F. Somers. Experiences using Intelligent Agent Technologies as a Unifying Approach to Network and Service Management. Proceedings of IS&N 98, Antwerp, Belgium. 1998.
17. G. Lavender and D. Schmidt. Active Object: An object behavioural pattern for concurrent programming. In J.M. Vlissides, J.O. Coplien, and N.L. Kerth, Eds. Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1996.
18. D. Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison Wesley, Reading, MA, 1997.
19. P. Maes. Agents that reduce work and information overload. Comm. of ACM, 37(7):30-40. 1994.
20. D.L. Martin, A.J. Cheyer and D.B. Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. Applied Artificial Intelligence 13:91-128. 1998.
21. B. Meyer. Object Oriented Software Construction, 2nd Ed. Prentice Hall, 1997
22. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF - The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, Eds. – Proc. 2nd Int. Workshop Mobile Agents (MA '98), Lecture Notes in Computer Science, 1477, pp. 50-67, Springer, Stuttgart, Germany, 1998.
23. FIPA-OS Home Page. Available at http://www.nortelnetworks.com/products/announcements/fipa/index.html.
24. H.S. Nwana. Software Agents: An Overview. The Knowledge Engineering Review, 11(3):205-244, 1996.

25. 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.
26. Object Management Group. 95-11-03: Common Services. 1997. Available at http://www.omg.org.
27. R.S. Patil, R.E. Fikes, P.F. Patel-Scheneider, D. McKay, T. Finin, T. Gruber 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.
28. A.S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In Proc. of the First Int. Conf. On Multi-Agent Systems, pp. 312-319, San Francisco, CA, 1995.
29. Reticular Systems. Agent Construction Tools. 1999. Available at http://www.agentbuilder.com.
30. Reticular Systems. AgentBuilder - An integrated Toolkit for Constructing Intelligence Software Agents. 1999. Available at http://www.agentbuilder.com.
31. J.R. Searle. Speech Acts: An Essay in the Phylosophy of language. Cambridge University Press, 1970.
32. Y. Shoham. Agent-oriented programming. Artificial Intelligence, 60(1):51-92. 1993.
33. Munindar P. Singh. Write Asynchronous, Run Synchronous. IEEE Internet Computing, 3(2):4-5. 1999.
34. K. Sycara, A. Pannu, M. Williamson and D. Zeng. Distributed Intelligent Agents. IEEE Expert, 11(6):36-46. 1996.
35. S.R. Thomas. The PLACA Agent Programming Language. In M.J. Wooldrige & N.R. Jennings (Eds.), Lecture Notes in Artificial Intelligence, pp. 355-370. Springer-Verlag, Berlin. 1994.
36. M. Wooldrige and N.R. Jennings. Intelligent Agents: Theory and Practice, The Knowledge Engineering Review, 10(2):115-152, 1995.