An Object-Oriented Framework to Realize Agent Systems

F. Bellifemine
CSELT, Torino, Italy
bellifemine@cselt.it

A. Poggi, G. Rimassa, P. Turci
Dipartimento di Ingegneria dell'Informazione
Università di Parma, Parma, Italy
{poggi,rimassa,turci}@ce.unipr.it

Abstract

In this paper, we present an agent model and platform we defined and implemented to realize efficient and reusable agent software through an agent development environment called 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 uses an agent model and a Java implementation that offer a good runtime efficiency and software reuse. Such an agent model is more "primitive" than the agent models offered by other systems, but such models can be implemented on the top of our " primitive" agent model. JADE agent platform tries to optimize 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.

1. Introduction

Agent-based technologies represent one of the most promising technological paradigms, however, they 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.

A lot of organizations are working towards the standardization of agent technologies starting from the work done by the Knowledge Sharing Effort [13] In this respect, FIPA (Foundation for Intelligent Physical Agents) represents one of the most interesting answers to the need for standards [4].

The standardization work of FIPA is in the direction to allow an easy interoperability between agent systems, because FIPA, beyond an agent communication language, specifies the key agents necessary for the management of an agent system and the ontology necessary for the interaction between two systems.

The output documents of FIPA 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 identifying the roles of some key agents necessary for the management of the platform, that is, the Agent Management System (AMS), the Agent Communication Channel (ACC) and the Directory Facilitator (DF).

Besides specifying other aspects as, for example, the agent-software integration, agent mobility and security, ontology service, and the human-agent communication, FIPA also specifies the Agent Communication Language (ACL) The syntax of the ACL is very close to the widely used communication language KQML [3]. The standard does not set out a specific mechanism for the transportation of messages. Since different agents might run on different platforms and use different networking technologies, the messages are encoded in a textual form.

A lot of people is involved in the realization of development environments to build agent systems (see, for example, AgentBuilder [16], dMARS [14], MOLE [20], the Open Agent Architecture [9], RETSINA [21] and Zeus [11]). Such development environment provide some predefined agent models and tools to ease 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] or following a standard as, for example, FIPA. However, none of them can be used to realize efficient and reusable agent software because they offer some specific agent architectures that must be used to realize all the agents of the system even if some of them must perform simple, primitive tasks which do not justify the complexity of the architecture used and because other agents execute tasks for which some other agent architectures are more suitable.

In this paper, we present an agent development environment called JADE. JADE (Java Agent Development Environment) that 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 agent platform tries to optimize 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.

In the next section, we introduce related work on agent construction tools. Section three presents the JADE agent platform. Section four describes JADE agent model. Finally, section five concludes discussing JADE features, giving a brief description of some applications realized with JADE and discussing the relationships between the JADE and the other agent software frameworks introduced in the paper.

2. Related work

A lot of research and commercial organizations are involved in the realization of agent applications and a considerable number of agent construction tools has been realized [15]. Some of the most interesting are AgentBuilder [16], MOLE [20], the Open Agent Architecture [9], RETSINA [21] and Zeus [11].

AgentBuilder [16] 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, analyzing the domain of agent operations, defining, implementing and testing agent software. 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 [18] agent model. Agents usually communicate through KQML messages; however, the developer has the possibility to define new communication commands to cope with her/his particular needs.

dMARS [14] is an agent-oriented development and implementation environment for building distributed system based on the BDI agent model offering support for system configuration, design, maintenance and re-engineering. Such a development environment has been successfully used to realize application in the fields of air traffic control and of telecommunication and business process management.

MOLE [20] is an agent system developed in Java whose agents do not have a sufficient set of features to be considered truly agent systems [6,22]. However, MOLE is important because it offers one of the best solution to support agent mobility. Mole agents are multi-thread entities identified by a globally unique agent identifier. Agents interact through two types of communication through RMI in the case of client/server interactions and message exchange in the case of peer-to-peer interactions.

The Open Agent Architecture [9] is a truly open architecture to realize 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. However, all the other agents must communicate via the facilitator that can become the bottleneck of the application.

RETSINA [21] offers reusable agents to realize applications. Each agent has four reusable modules for communicating, planning, scheduling and monitoring the execution of tasks and requests from other agents. RETSINA agents communicate through KQML messages. However, agents developed by others and non-agentized software can inter-operate with RETSINA agents by building some specialized gateway agents (one for each non-RETSINA agent or non-agentized software system) handling communication channels, different data and message formats, etc. RETSINA provides three kinds of agent: i) interface agents to manage the interaction with users; ii) task agents to help users in the execution of tasks; and iii) information agents to provide intelligent access to heterogeneous collections of information sources.

Zeus [11] allows the rapid development of Java agent systems by providing a library of agent components, 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, organizational layer, coordination layer and communication layer. The API layer allows the interaction with non-agentized world. The definition layer manages the task the agent must perform. The organizational layer manages the knowledge about the other agents. The coordination layer manages coordination and negotiation with other agents. Finally, the communication layer allows the communication with the other agents.

3. 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 [7]. The goal of JADE is to simplify development

while ensuring standard compliance through a comprehensive set of system services and agents.

The JADE Agent Platform 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.

JADE 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. 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 delivering runtime environment support to some JADE agents. Java RMI is used to communicate among the containers and each one of them can also act as an IIOP client 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. The two mandatory system agents, that is the AMS and the default DF, run within the front-end container.

New agent containers can be added to a running JADE platform; as soon as a non-front-end container is started, it follows a simple registration protocol with the front-end and adds itself to a container table maintained by the front-end. Users can employ simple command line options to tell on which host and port the front end is listening for new container registrations.

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 client/server and reactive systems. Clearly, there is some gap to bridge in order to map an explicitly-addressed message-passing model such as the FIPA ACL one into the connection oriented, request/response based communication model of distributed objects. This is why in JADE ordinary agents are not distributed objects, but agent containers are.

A more detailed description is needed of the various means JADE has to forward ACL messages and of the way JADE selects among them, to better clarify this important design issue. A software agent, in compliance to FIPA agent model, has a globally-unique identifier (GUID), that can be used by every other agent or software entity to address it with ACL messages. Since a FIPA97 GUID resembles an email address, it has the form: <agent name> @ <platform address> (where platform address can be either the IIOP URL or the stringified CORBA IOR for the agent platform, seen as an IIOP reachable CORBA object), it is fairly easy to recover the agent name and the platform address from it. JADE then compares the receiver platform address with its own platform address: if they differ, the receiver resides on some other platform, possibly a non-JADE one, and standard IIOP messaging is to be used. So the ACLMessage Java object representing the actual message is converted into a String and sent on an IIOP channel created with receiver's platform.

Otherwise, if the receiver and the sender reside on the same agent platform, JADE uses event dispatching when the two agents are within the same container and Java RMI when they are on different containers of the same agent platform. When Java events are used, the ACLMessage object is simply cloned and passed to the receiver agent; when using RMI, instead, the ACLMessage object is serialised and unserialised transparently by RMI runtime.

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 new agents, shutting down peripheral containers, etc.) through the AMS. The Directory Facilitator agent also has a GUI of its own, with which the DF can be administered, adding or removing agents and configuring their advertised services.

The two graphical tools with which JADE users can debug their agents are the Dummy Agent and the Sniffer Agent. The Dummy Agent is a simple yet very useful tool for inspecting message exchanges among agents and provides a graphical interface to edit, compose and send ACL messages to agents, to receive and view messages from agents, and, eventually, to save/load messages to/from disk. The Sniffer Agent allows to track messages exchanged in a JADE agent platform and provides a graphical interface to display the messages exchanged among a group of agents using a notation similar to UML Sequence Diagrams.

4. JADE agent model

FIPA specifications state nothing about agent internals: this was an explicit choice of the standard consortium, according to the opinion that interoperability can be granted only by mandating external agent behaviour through standard ACL, protocols, content languages and

ontology, allowing platform implementers to take different design and implementation paths. While this is correct in principle, when a concrete implementation as JADE must be designed and built there are many more issues to address. Quite important among them is how agents are executed within their platform; this does not impair interoperability because different agents communicate only through message exchanges. Nevertheless, the execution model for an agent platform is a major design issue which both affects runtime performance and imposes specific programming styles on application agent developers. Various competing forces need to be taken into account when addressing such a problem; as will be shown in the following, JADE solution stems from a careful balancing of these forces. Some of them come from software engineering guidelines, whereas others derive from theoretical agent properties and are then peculiar of intelligent agents systems.

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. Actions performed by an agent don't just depend on received messages but also on internal state and accumulated history of a software agent.

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.

The FIPA standard bases its *Agent Communication Language* (*ACL*) on *speech-act theory* [17] and uses a mentalistic model to build a formal semantic for the various kinds of messages (*performatives*) agents can send to each other. This approach is quite different from the client/server one followed by distributed object systems and rooted in *Design by Contract* [10], where *invocations* are made on exported *interfaces*. A fundamental difference is that, while invocations on interfaces can either succeed or fail, a speech act *request*, besides *agree* and *failure*, can also get back a *refuse* performative, expressing the unwillingness of the receiver agent to perform the requested action.

Trying to map the aforementioned intrinsic agent properties into concrete design decisions, the following requirement/design mapping list was produced.

- *Agents are **autonomous** then are **active objects***
- *Agents are **social** then **intra-agent concurrency** is needed*
- *Agent 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 software agent to be an *active object* [8]; each agent must have at least a Java thread in order to be able to proactively start new conversations and also to make plans and pursue goals. The abstract need for sociality has the practical outcome of allowing an agent to engage in multiple conversations simultaneously; so a software agent must deal with a significant amount of concurrency.

The third requirement suggests asynchronous message passing as an implementable way to represent an information exchange between two autonomous, independent entities that also has the added benefit of producing more reusable interactions [19]. Similarly, the last requirement stresses that in a Multi Agent System the sender and the receiver have equal rights (as opposed to client/server systems where the receiver is supposed to obey the sender). Besides being able to say "no", an autonomous agent must also be allowed to say "I don't care", ignoring a received message as long as he wishes. This added need rules out purely reactive message handlers and advocates using a *pull consumer* messaging model instead [12], where incoming messages are buffered until their receiver decides to read them.

The abstraction used to model agent tasks is the *Behaviour*: each JADE agent holds a collection of behaviours which are scheduled and executed to carry on agent duties (see figure 3). Behaviours represent logical threads of a software agent implementation. According to *Active Object* design pattern [8], every JADE agent runs in its own Java thread, thereby satisfying autonomy property; instead, in order to keep small the number of 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.

The main advantage of using a single Java thread for all agent behaviours lies in greatly reduced multithreading overhead. Recalling the three major costs paid to enjoy multithreading benefits, one can see that with JADE model thread creation and deletion happens rarely, because agents are rather long lived software objects. Besides, synchronisation between different threads is not even needed, since different agents share no common environment. Thus, only the third cost, i.e., thread scheduling, remains; this is the least among the three and could only be avoided by making the whole agent platform single-threaded. This would be, in our opinion, an unacceptable limitation: according to JADE model, an agent platform is still a multithreaded execution environment, with Java threads used as coarse grained concurrency units and cooperative behaviours providing a finer grain of parallelism.

Sometimes real intra-agent multithreading may seem unavoidable: for example, an agent acting as a wrapper onto a DBMS could issue multiple queries in parallel, or an agent might want to block on a stream or socket while still being able to engage in ordinary conversations. Really, this kind of problems occur only when an agent must interact with some non-agent software; FIPA

acknowledges that these are boundary conditions for the execution model and deals with them in a separate part of the standard (namely, FIPA part 1 is about agent management and FIPA part 3 deals with external, non-agent software).

JADE *thread-per-agent* execution model can deal alone with all the most common situations involving only agents: this is because every JADE agent owns a single message queue from which all ACL messages are to be retrieved. Having multiple threads but a single mailbox would bring no benefit in message dispatching, since all the threads would still have to synchronise on the shared mailbox. 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, as long as appropriate synchronisation is used.

The developer who wants to implement an agent must extend the *Agent* class and implement the agent-specific tasks by writing one or more *Behaviour* subclasses, instantiate them and add the behaviour objects to the agent. User defined agents inherit from the *Agent* class the basic capability of registering and deregistering with their platform and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send and receive ACL messages, use standard interaction protocols, register with several domains).

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 [5]. JESS is a scripting environment for rule programming written in Java offering an engine using the Rete algorithm to process rules. Therefore, while JADE provides the shell of the agent and guarantees the FIPA compliance, JESS allows using rule-oriented programming to define agent behaviours and exploiting its engine to execute them.

5. 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 can 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 [2] 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 [1] 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 [5] as agent reasoning engine.

The work has been partially supported by a grant from CSELT, Torino.

12. References

[1] The CoMMA Project Home Page: http://www.ii.atos-group.com/sophia/comma/HomePage.htm.
[2] The FACTS Project Home Page: http://www.labs.bt.com/profsoc/facts/.
[3] 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.
[4] FIPA - Foundation for Intelligent Physical Agents. http://www.fipa.org.
[5] E.J. Friedman-Hill. Jess, The Java Expert System Shell. Sandia National Laboratories, Livermore, CA. 1999. http://herzberg1.ca.sandia.gov/jess/.
[6] M.R. Genesereth and S.P. Ketchpel. Software Agents. Comm. of ACM, 37(7):48-53.1994.
[7] The JADE Project Home Page: http://sharon.cselt.it/projects/jade.
[8] 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.
[9] D.L. Martin, A.J. Cheyer and D.B. Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. Applied Artificial Intelligence. 1998.
[10] B. Meyer. Object Oriented Software Construction, 2nd Ed. Prentice Hall, 1997.
[11] H.S. Nwana, D.T. Ndumu and L.C. Lee. ZEUS: An advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. In: Proc of PAAM98, pp. 377-391, London, U.K., 1998.
[12] Object Management Group. 95-11-03: Common Services. 1997. http://www.omg.org.
[13] 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.
[14] A.S. Rao and M.P. Georgeff. BDI agents: from theory to practice. In Proc. of 1st Int. Conf. On Multi-Agent Systems, pp. 312-319, San Francisco, CA, 1995.
[15] Reticular Systems. Agent Construction Tools. 1999. http://www.agentbuilder.com.
[16] Reticular Systems. AgentBuilder - An integrated Toolkit for Constructing Intelligence Software Agents. 1999. http://www.agentbuilder.com.
[17] J.R. Searle. Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, 1969.
[18] Y. Shoham. Agent-oriented programming. Artificial Intelligence, 60(1):51-92. 1993.
[19] M.P. Singh. Write Asynchronous, Run Synchronous. IEEE Internet Computing, 3(2):4-5. 1999.
[20] M. Straßer, J. Baumann and F. Hohl. Mole - A Java based Mobile Agent System. In: M. Mühlhäuser: (ed.), Special Issues in Object Oriented Programming. dpunkt Verlag, pp. 301-308, 1997.
[21] K. Sycara, A. Pannu, M. Williamson and D. Zeng. Distributed Intelligent Agents. IEEE Expert, 11(6):36-46. 1996.
[22] M. Wooldrige and N.R. Jennings. Intelligent Agents: Theory and Practice, The Knowledge Engineering Review, 10(2):115-152, 1995.