SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:103–128

Developing multi-agent systems with a FIPA-compliant agent framework

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

[1]*CSELT S.p.A., Via G. Reiss Romoli 274, Torino I-10148, Italy*
[2]*Dipartimento di Ingegneria dell'Informazione, University of Parma, Parco Area delle Scienze 181A, Parma I-43100, Italy*

SUMMARY

To ease large-scale realization of agent applications there is an urgent need for frameworks, methodologies and toolkits that support the effective development of agent systems. Moreover, since one of the main tasks for which agent systems were invented is the integration between heterogeneous software, independently developed agents should be able to interact successfully.

In this paper, we present JADE (Java Agent Development Environment), a software framework to build agent systems for the management of networked information resources in compliance with the FIPA specifications for inter-operable 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. JADE can then be considered to be an agent middle-ware that implements an efficient agent platform and supports the development of multi-agent systems. It deals with all the aspects that are not peculiar to agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle management. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: agent development framework; agent platform; distributed systems; FIPA; Java

INTRODUCTION

Every day, developers have the problem of constructing ever larger and more complex software applications. Developers are now creating enterprise-wide and world-wide applications that must operate across corporations and continents and inter-operate with other heterogeneous systems. Such applications are difficult to produce with traditional software technologies because of the limits of these

*Correspondence to: Agostino Poggi, Dipartimento di Ingegneria dell'Informazione, University of Parma, Parco Area delle Scienze 181A, Parma I-43100, Italy.
†E-mail: poggi@ce.unipr.it

Contract/grant sponsor: CSELT, Torino, Italy

Copyright © 2001 John Wiley & Sons, Ltd.

Received 12 July 1999
Revised 5 June 2000 and 25 August 2000
Accepted 26 August 2000

technologies in coping with dynamically changing and unmanaged environments. It would appear that agent-based technologies represent a promising tool for the deployment of such applications because they offer the high-level software abstractions needed to manage complex applications and because they were invented to cope with distribution and inter-operability [1–6].

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 there are standards to support agent inter-operability and adequate environments for the development of agent systems. In this period, there is a lot of activity concerning the development of agent systems (see, for example, DMARS [7], RETSINA [8] and MOLE [9]), and the standardization of these systems, where in the past KSE [10], OMG [11] and then FIPA [12] led the activity.

However, to date there is no system accepted as being considerably superior to the others. At this point in time, the FIPA standardization effort is predominant and more and more systems based on the FIPA are available.

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 (KQML [13]). However, a shared communication language is not the only element required to support inter-operability between different agent systems; common agent services and ontologies are also needed. The standardization work of FIPA acknowledges this issue and, in addition to 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 Environment), a software framework to write agent applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems. The next two sections respectively introduce the FIPA specifications and related work on software frameworks to develop agent systems. The following sections describe the main features of JADE: the architecture of the agent platform, the communication subsystem and the agent model, and discuss the main features of the current implementation of JADE, and its use to develop applications and to perform inter-operability tests with other agent platforms. Finally, the last section gives a brief comparison between JADE and the other agent software frameworks and presents the new features that will be added.

FIPA SPECIFICATIONS

The Foundation for Intelligent Physical Agents (FIPA) [12] is an international non-profit association of companies and organizations sharing the effort to produce specifications for generic agent technologies. FIPA does not promote a technology for just 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 inter-operability.

The FIPA standardization process relies on two main assumptions. The first is that the time required to reach a consensus and to complete the standard should be as short as possible, should not impede progress, but should act as a promoter of stronger industrial commitment in agent technology. The second assumption is that only the external behaviour of system components should be specified,

Copyright © 2001 John Wiley & Sons, Ltd.

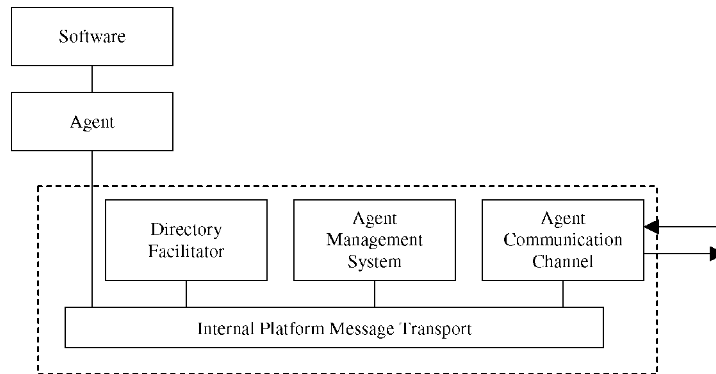

Figure 1. FIPA reference model of an agent platform.

leaving implementation details and internal architectures to platform 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, 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, as shown in Figure 1. 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 for 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 passes on yellow page services to the agent platform. Notice that no restriction is given to the actual technology used for platform implementation: an e-mail based platform, a CORBA based one, a Java multi-threaded application, etc. could all be FIPA compliant implementations.

According to the FIPA definition [12], an agent is the fundamental actor in a domain. It is capable of bringing together a number of service capabilities to form a unified and integrated execution model which can include access to external software, human users and communication facilities.

Of course, the specifications also define the Agent Communication Language (ACL). Agent communication is based on message passing, where agents communicate by sending individual messages to each other. The FIPA ACL is a standard message language and sets out the encoding, semantics and pragmatics of the messages. It 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. 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

Copyright © 2001 John Wiley & Sons, Ltd.

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 FIPA ACL which should eliminate any ambiguity and confusion from the usage of the language.

The FIPA ACL has three important and interesting features.

1. It is independent of the actual content, since it only defines the communicative intention under the transmission of the message.
2. Its formal semantic element is defined in terms of the feasibility preconditions and the rational effect that allows a communicative act to be scheduled and planned as a normal action (e.g. opening the door); in addition, the semantics allows an agent to consider a message in an explicit manner, if and when needed.
3. The ACL provides the bases for the specification of interaction protocols and common patterns of conversation between agents aimed at specifying high-level tasks, such as delegating a task, negotiating conditions, and some forms of auctions.

FIPA 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 range from simple query and request protocols, to more complex ones, such 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 authors consider that the strength of FIPA is to be found in its approach: not just the specification of an ACL but also of all those services and infrastructures needed to allow existence and management of agents; not a standard for a single application, but a standard that tries to take care of the requirements of different applications, ranging from the TV entertainment domain to network management and nomadic applications. The interested reader should refer directly to the FIPA Web page [12].

RELATED WORK

A lot of research and commercial organizations are involved in the creation of agent applications, and a considerable number of agent construction tools have been produced [14]. Among those of major interest are AgentBuilder [15], ASL [16], Bee-gent [17], FIPA-OS [18], Grasshopper-2 [19], MOLE [9], the Open Agent Architecture [20], RETSINA [8] and Zeus [21].

AgentBuilder [15] 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, analysing the domain of agent operations, and defining, implementing and testing agent software. The Run-Time System provides an agent engine, i.e. an interpreter, used as an execution environment of agent software. AgentBuilder agents are based on a model derived from the Agent-0 [22] and PLACA [23] agent models. Agents usually communicate through KQML messages; however, the developer has the option to define new communication commands in order to cope with her/his particular needs.

Copyright © 2001 John Wiley & Sons, Ltd.

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

Bee-gent [17] is a software framework to develop agent systems compliant with FIPA specifications produced by Toshiba. Such a framework provides two types of agents: wrapper agents used to agentify existing applications and mediator agents supporting the wrapper coordination by handling all their communications. Agents communicate through XML/ACL messages and mediator agents are mobile agents that can migrate around the network by themselves. Bee-gent also offers: (i) a graphic RAD tool to describe agents through state transition diagrams; (ii) naming/directory facilities to locate agents, databases and applications; (iii) ontology facilities to translate words referring to the same entity; and (iv) security and safety facilities based on digital fingerprint authentication and secret key encryption in order to prevent the mediator agents from being tampered with, or wiretapped while moving around the network.

FIPA-OS [18] is another software framework to develop agent systems compliant with FIPA specifications that has been created by NORTEL. Such a framework provides the mandatory components that produce 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 by taking advantage of FIPA-OS agent platforms.

Grasshopper-2 [19] is a pure Java based Mobile Agent platform, conforming with existing agent standards, as defined by the OMG-MASIF (Mobile Agent System inter-operability Facility) [11] and FIPA specifications. Thus Grasshopper-2 is an open platform, enabling maximum inter-operability with other mobile and intelligent agent systems, and easy integration of existing and upcoming CORBA and Java services and APIs. The Grasshopper-2 environment comprises 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. The life of an agent system is based on the services offered by a core agency. A core agency provides only those capabilities that are absolutely crucial for the execution of agents. Agents access the core agency for retrieval of information about other agents, agencies or places, or in order to move to another location. Users are able to monitor and control all activities within an agency by accessing the core services, i.e. Communication Service, Registration Service, Transport Service, Security Service, and Management Services. Moreover, Grasshopper-2 provides a Graphic User interface for user-friendly access to all of the functions of an agent system.

MOLE [12] is an agent system developed in Java whose agents do not have a sufficient set of features to be considered truly agent systems [2,3]. 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: via RMI for client/server interactions and through message exchanges for peer-to-peer interactions.

The Open Agent Architecture [20] is a truly open architecture used to create 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, and this can produce an application bottleneck.

Copyright © 2001 John Wiley & Sons, Ltd.

RETSINA [8] offers reusable agents to create 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. However, agents developed by others and non-agentized software can inter-operate with RETSINA agents by building some 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 [21] 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, organizational layer, coordination layer and communication layer. The API layer allows interaction with the non-agentized world. The definition layer manages the task the agent must perform. The organizational layer manages the knowledge concerning other agents. The coordination layer manages coordination and negotiation with other agents. Finally, the communication layer allows the communication with other agents.

JADE

JADE (Java Agent Development framework) is a software framework to aid the development of agent applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems. JADE is an Open Source project, and the complete system can be downloaded from the JADE Home Page [24]. The purpose 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.

1. 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 (see Section '*Distributed agent platform*' for details).
2. Distributed agent platform. The agent platform can be split onto several hosts. Usually, 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 still be executed by one agent, and JADE schedules these tasks in a cooperative way (see Section '*Distributed agent platform*' for details).
3. 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 (see Section '*Distributed agent platform*' for details).

Copyright © 2001 John Wiley & Sons, Ltd.

4. Java API to send/receive messages to/from other agents; ACL messages are represented as ordinary Java objects (see Section '*From agent theory to class design*' for details).
5. FIPA97-compliant IIOP protocol to connect different agent platforms (see Section '*Message delivery subsystem*' for details).
6. Light-weight transport of ACL messages within the same agent platform, because the messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures (see Section '*Message delivery subsystem*' for details).
7. Library of FIPA interaction protocols ready to be used (see Section '*Interaction Protocols*' for details).
8. Graphic user interface to manage several agents 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 (see Section '*Tools for platform management and monitoring*' for details).

The JADE system can be described from two different points of view. On the one hand, JADE is a run-time 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 the current strengths and weaknesses of JADE and will describe the future improvements envisaged in the JADE development roadmap.

JADE RUN-TIME SYSTEM ARCHITECTURE

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

Distributed agent platform

The JADE Agent Platform complies with the FIPA97 specifications and includes all the system agents that manage the platform; 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 the Java run-time 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 of them acting as a front end for inter-platform IIOP communication. A JADE system comprises one or more *Agent Containers*, each living in a separate Java Virtual Machine and delivering run-time 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

Copyright © 2001 John Wiley & Sons, Ltd.

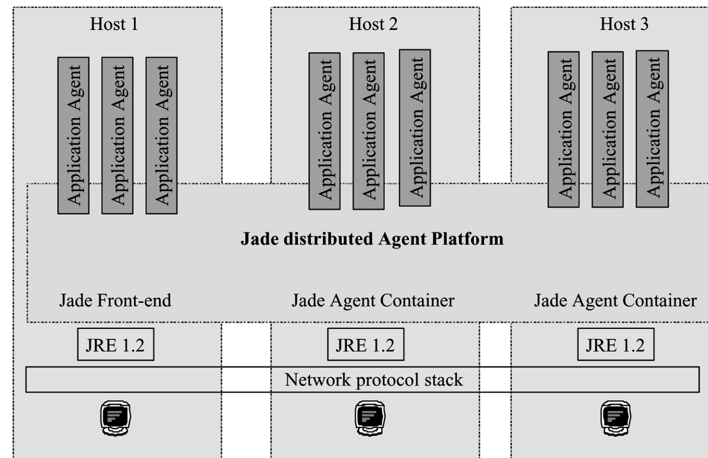

Figure 2. Software architecture of one agent platform.

platforms. A special, *Front End* container is also an IIOP server, listening to 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. Figure 2 shows the architecture of a JADE Agent Platform.

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.

The Java Remote Method Invocation API is used as the communication middle-ware that makes JADE take on the appearance of a single platform, even if it is distributed. Every agent container exports an RMI remote interface to provide a distributed objects infrastructure for platform operations, such as agent life cycle management. For example, when the AMS is asked to suspend an agent, it might often be the case that the agent is running on a different container, and this is unknown to all of the agents, AMS included, because agent-level addressing sees a JADE platform as a whole, following FIPA naming specifications. For this reason, the AMS can but trust the underlying JADE infrastructure to find the relevant agent. For this reason it simply calls a *suspendAgent()* method, passing the agent name as an argument. This method will produce a local call if the agent to suspend lives within the front end container or, if this is not the case, an RMI remote call.

Message delivery subsystem

The FIPA agent communication model is peer-to-peer, though a multi-message context is provided by interaction protocols and conversation identifiers. JADE, on the other hand, uses transport technologies such as RMI, CORBA and event dispatching which are typically associated with client/server

Copyright © 2001 John Wiley & Sons, Ltd.

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

A more detailed description is needed of the various methods that JADE can use to forward ACL messages and of the way JADE selects from among them in order to better clarify this important design issue. A software agent, in compliance with the 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. Likewise, an agent will put its GUID into the *:sender* slot of ACL messages it sends around to acknowledge itself as the message sender. So, JADE must figure out receiver location by simply looking at the *:receiver* message slot. Since an FIPA97 GUID resembles an e-mail 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), and so it is fairly easy to recover the agent name and the platform address. JADE then compares the receiver platform address with its own platform address. If they differ, the receiver resides on some other platform, possibly of non-JADE type, 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.

On the other hand, if the receiver and the sender reside on the same agent platform, JADE uses event dispatching when the two agents are in the same container and Java RMI when they are in 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, on the other hand, the *ACLMessage* object is serialized and unserialized transparently by RMI run time.

Therefore, when an ACL message is sent to a software agent, there are three possibilities.

1. *Receiver in the same container of the same platform:* Java events are used; the cost is a cloning of the *ACLMessage* object and a local method call.
2. *Receiver in a different container of the same platform:* Java RMI is used, the cost is a message serialization on the sender side, a remote method call and a message unserialization on the receiver side.
3. *Receiver on a different platform:* CORBA IIOP is used; the cost is the conversion of the *ACLMessage* object into a *String* object and an IIOP marshalling on the sender side, a remote method call and an IIOP unmarshalling followed by ACL parsing on the receiver side.

Address management and caching

As discussed above, JADE tries to select the most convenient of these three transport mechanisms in accordance with the location of the sender and the receiver. This opens up the issue of where the knowledge about which agents are on which containers is kept. 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 the container where the agent lives. When a container has to discover exactly where an agent of its platform is located, it looks first into its LADT, and then, if the search fails, asks the front-end for the container through a GADT lookup. Since looking up the GADT requires a remote

Copyright © 2001 John Wiley & Sons, Ltd.

RMI invocation, JADE uses an address caching technique to avoid querying the front-end continuously for address information. JADE address caching is transparent and completely orthogonal to message transport. For this reason local, RMI and IIOP addresses are all cached in the same way and on cache hits the agent container does not even need to know the specific kind of address it is using.

This should support agent mobility. An address referring to a mobile agent can change its nature (e.g. from local to RMI) over time. Transparent caching means that a messaging subsystem will not be affected when agent mobility is introduced into JADE. Moreover, if new remote protocols are needed in JADE (e.g. a wireless protocol for nomadic applications), they will be seamlessly integrated inside the messaging and address caching mechanisms. The JADE cache replacement policy is of the standard LRU type, and a stale cached address is not invalidated until it is used, following an optimistic attitude. When a container tries to use a stale address, local or remote, it receives an exception and refreshes the cache item contacting the front end (for RMI addresses) or looking up its own LADT (for local addresses).

Tools for platform management and monitoring

In addition to a run-time 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; they simply rely on JADE AMS. The general management console for a JADE agent platform is called RMA (*Remote Management 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. On the one hand, the RMA asks the AMS to be notified about the changes of state of platform agents; on the other hand, it transmits to the AMS the requests for creation, deletion, suspension and restart received by the user. 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. The Dummy Agent facilitates validation of an agent message exchange pattern before its integration into a multi-agent system and facilitates interactive testing of an agent. The graphic interface provides support 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 makes it possible to track messages exchanged in a JADE agent platform. When the user decides to sniff a single agent or a group of agents, every message directed to or coming from that agent or group is tracked and displayed in the sniffer window, using a notation similar to UML *Sequence Diagrams*. Every ACL message can be examined by the user, who can also save and load every message track for later analysis.

JADE AGENT DEVELOPMENT MODEL

FIPA specifications state nothing about agent internals: this was an explicit choice of the standard consortium, following the opinion that inter-operability can be granted only by mandating external

Copyright © 2001 John Wiley & Sons, Ltd.

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 such as JADE has to be designed and built there are many more issues to be addressed. One of the most important of these issues is how agents are executed within their platform. This does not impair inter-operability because different agents communicate only through message exchanges. Nevertheless, the execution model for an agent platform is a major design issue which affects both run-time 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, the JADE solution stems from a careful balancing of these forces. Some of these come from ordinary software engineering guidelines, whereas others derive from theoretical agent properties and are therefore peculiar to intelligent agent systems.

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 is also able to start new communicative acts of its own. Actions performed by an agent do not just depend on received messages but also on the 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 and can even develop an overall strategy together with its peers.

The FIPA standard bases its *Agent Communication Language (ACL)* on *speech-act theory* [25] 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 type followed by distributed object systems and rooted in *Design by Contract* [26], 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 receive 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:

Theoretical requirement		**Design solution**
Agents are autonomous	⇒	**Agents are active objects**
Agents are social	⇒	**Intra-agent concurrency is needed**
Messages are speech acts	⇒	**Asynchronous messaging has to be used**
Agents can say 'no'	⇒	**Peer-to-peer communication model is needed**

The autonomy property requires each software agent to be an *active object* [27]; each agent must have at least one Java thread in order to be able to proactively start new conversations and 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 independent entities that also has the additional benefit of producing more reusable interactions [28]. Similarly, the last requirement stresses that in a multi-agent system the sender and the receiver have equal rights (which is not the case with client/server systems

Copyright © 2001 John Wiley & Sons, Ltd.

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 for as long as it wishes. This added requirement rules out purely reactive message handlers and advocates using a *pull consumer* messaging model [29], where incoming messages are buffered until their receiver decides to read them.

JADE agent concurrency model

The above considerations make it easier to decide 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. It is possible that it will be necessary to have a thread for every conversation the agent gets involved with, and maybe even more threads to perform agent reasoning.

Unfortunately, current hardware and operating systems have clear limits with respect to the maximum number of threads that can be run effectively on a system. To achieve run-time efficiency, the various costs associated with multi-threading must be evaluated. In decreasing cost order, they are:

1. *thread creation and deletion;*
2. *synchronization between threads;*
3. *thread scheduling.*

Additional design forces result from a software engineering viewpoint. JADE can be seen as a software framework for agent-based applications, with application programmers concentrating on writing agents and on specifying each agent role within the agent society. Since complex tasks in multi-agent systems are usually tackled using collaboration among many agents, a single agent is typically a strongly cohesive piece of software. On the other hand, asynchronous message passing with a pull consumer messaging model leads to a very loose coupling between different agents. Furthermore, no implementation inheritance (and no code reuse) is considered when dealing with software agents. In this way, software agents have a strong resemblance with actors, and indeed the JADE execution model has its roots in actor languages.

The abstraction used to model agent tasks is called *Behaviour*: each JADE agent holds a collection of behaviours which are scheduled and executed to perform agent duties (see Figure 3). Behaviours represent the logical threads of a software agent implementation. According to the *Active Object* design pattern [27], every JADE agent runs in its own Java thread, thereby satisfying autonomy. On the other hand, in order to reduce 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 the considerable reduction in multi-threading overhead. Recalling the three major costs that have to be paid to enjoy multi-threading benefits, one can see that with the JADE model thread creation and deletion is a rare occurrence, because agents are long-life software objects. Besides, synchronization between different threads is not even needed, because different agents have no common environment. Thus, only the third cost, i.e. thread scheduling, remains. This is the least important of the three and could only be avoided by making the entire agent platform single-threaded. This would be, in our opinion, an unacceptable limitation. With the JADE model, an agent platform is still a multi-threaded execution environment,

Copyright © 2001 John Wiley & Sons, Ltd.

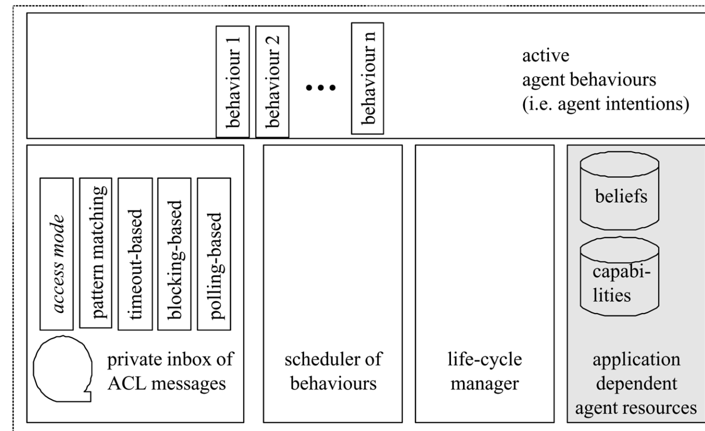

Figure 3. JADE agent architecture.

with Java threads used as coarse grained concurrency units and cooperative behaviours providing a finer grain of parallelism.

Adopting a *thread-per-behaviour* execution model would incur significant synchronization costs (a Java *synchronized* method is about 100 times slower than an ordinary method) because all behaviours share a common representation of agent internal state. Moreover, it is often the case that an agent creates new behaviours on demand (for example, agents acting as information brokers usually start new conversations as soon as a request message is received). For that kind of agent, *thread-per-behaviour* would also have significant thread creation overheads (unless some form of thread pooling is implemented, which would impose a significant amount of work on platform implementers).

Using a single Java thread to handle multiple agent behaviours needs some sort of scheduling policy. JADE relies on a '*cooperative scheduling on top of the stack*', in which all agent behaviours are run from a single stack frame without context saving (*on top of the stack*) and a behaviour continues to run until it returns from its main function and cannot be pre-empted by other behaviours (*cooperative scheduling*). Of course ordinary pre-emption is still active between different agent threads and among JADE system threads: cooperative scheduling is strictly an intra-agent policy.

Using cooperative behaviours to model multiple agent conversations is a lightweight approach to concurrency, which tries to achieve low latency by working entirely in user space. Similar techniques are customary in modern high performance network protocols and messaging libraries [30,31]. The JADE model is also an attempt to provide fine grained parallelism on coarser grained hardware. A similar, stack based execution model is used by the Illinois Concert run-time system [32]. This system is able to provide a run-time environment for parallel object oriented languages. Concert can execute concurrent method calls optimistically on the stack, reverting to real thread spawning only when the method is about to block, thus saving the context for the current call only when forced to.

However, the Concert system relies on compiler support to select concurrency strategy (parallel or lazily stack based) and to save method context when needed. On the other hand, JADE is not a new

Copyright © 2001 John Wiley & Sons, Ltd.

language but a Java development framework: saving the context when an agent behaviour is blocking would put a heavy burden on JADE users, because they should write state saving code themselves, since no compiler would be going to write it for them.

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

Turn the method body into an object whose class inherits from Behaviour.
Turn method local variables into behaviour instance variables.
Add the behaviour object to agent behaviour list during agent startup.

The above guidelines apply the *reification technique* [36] to agent methods, in accordance with the *Command* design pattern [34,35]; an agent behaviour object reifies both a method and a separate thread executing it. While reification yields enhanced flexibility in task scheduling and execution, it often results in verbose program text (a problem which is also common in reflective code). A whole new class must be written and instantiated for each agent behaviour, and this can easily lead to programs that are difficult to understand and maintain. JADE application programmers can compensate for this shortcoming using Java *Anonymous Inner Classes*. This language feature is similar to Smalltalk code blocks and makes the amount of code necessary for defining an agent behaviour only slightly higher than for writing a single Java method.

Sometimes real intra-agent multi-threading 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. In reality, these types of problems occur only when an agent has to interact with 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 (FIPA part 1 concerns agent management and FIPA part 3 deals with external, non-agent software).

The JADE *thread-per-agent* execution model can deal alone with all of the most common situations involving only agents: this is because every JADE agent has 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 synchronize 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. For this reason, 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 synchronization is used.

Using behaviours to build complex agents

The developer who wants to build 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 up to implement the custom behaviour of the agent (e.g. send and receive ACL messages, use standard interaction

Copyright © 2001 John Wiley & Sons, Ltd.

protocols, register with several domains). Moreover, user agents inherit from their *Agent* superclass two methods: *addBehaviour(Behaviour)* and *removeBehaviour(Behaviour)*, which allow the managing of 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 (Figure 4 shows an annotated UML class diagram for JADE behaviours). For example, JADE offers a so-called *JessBehaviour* that allows full integration with JESS [36]. JESS is a scripting environment for rule based 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 exploits its engine to execute them.

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 has to return *true* when the behaviour has finished and can be removed from the queue, *false* when the behaviour has not yet finished and the *action()* method has to be executed again; the *reset()* method, used to restart a behaviour from the beginning.

Because of the non-pre-emptive multi-tasking model chosen for agent behaviours, agent programmers must avoid the use of endless loops and the performing of long operations within *action()* methods. This is because when a particular behaviour *action()* is running no other behaviour can proceed until the end of the method (of course this is only true for behaviours of the same agent: behaviours of other agents run in different Java threads and can still proceed independently).

Moreover, since no stack context is saved, every time the *action()* method is run from the beginning: there is no way to interrupt a behaviour in the middle of its *action()*, pass on the CPU to other behaviours and then start the original behaviour back from where it left off.

For example, suppose a particular operation *op()* is too long to be run in a single step and is therefore broken into three sub-operations, named *op1()*, *op2()* and *op3()*; to achieve the desired functionality one must call *op1()* the first time the behaviour is run, *op2()* the second time and *op3()* the third time, after which the behaviour must be marked as terminated. The code will be as follows:

```
public class my3StepBehaviour {
  private int state = 1;
  private boolean finished = false;
  public void action() {
    switch (state) {
      case 1: { op1(); state++; break; }
      case 2: { op2(); state++; break; }
      case 3: { op3(); state = 1; finished = true; break; }
    }
  }
  public boolean done() {
    return finished;
  }
}
```

Copyright © 2001 John Wiley & Sons, Ltd.

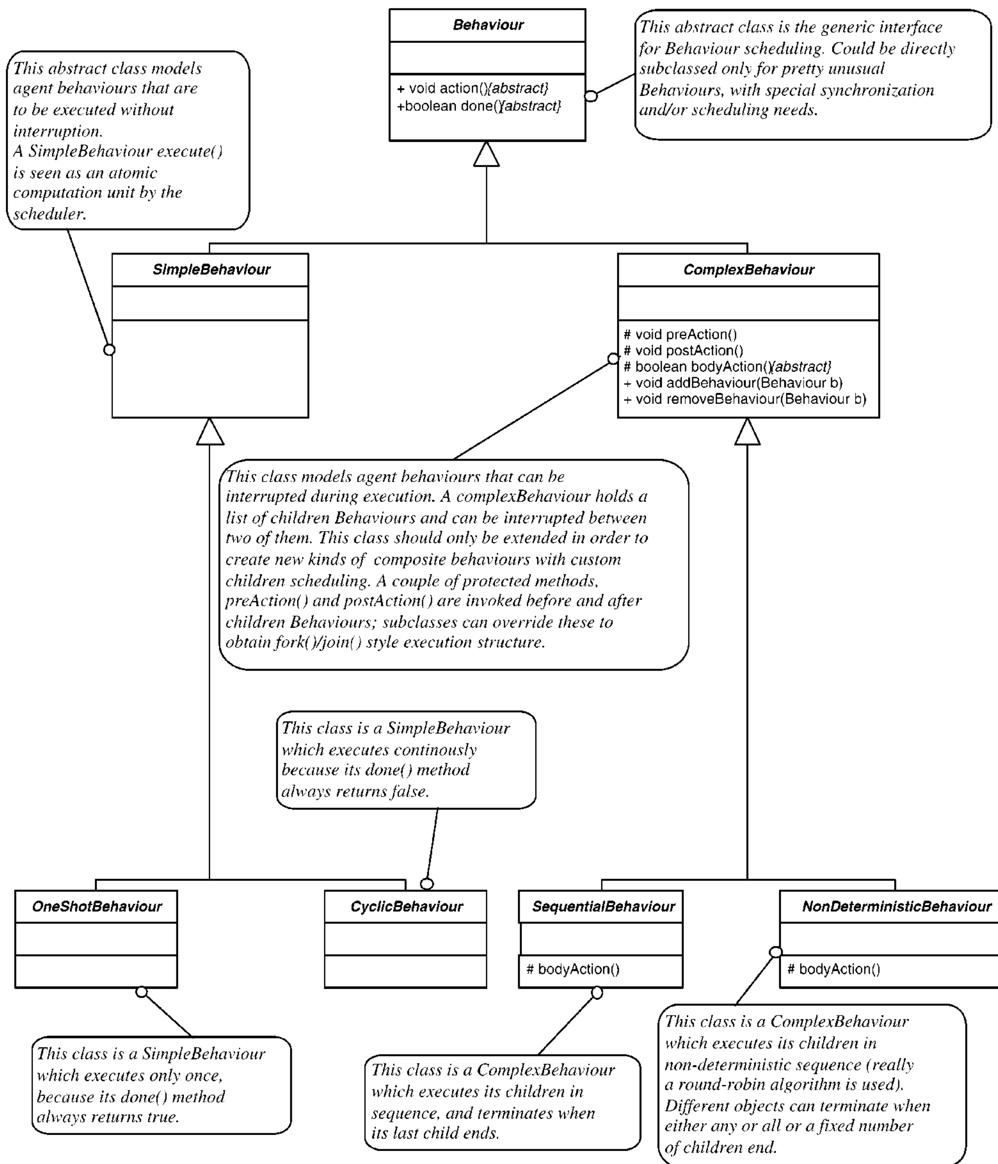

Figure 4. UML model of the behaviour class hierarchy.

Copyright © 2001 John Wiley & Sons, Ltd.

Following this idiom, agent behaviours can be described as finite state machines, keeping their overall state in their instance variables.

When dealing with complex agent behaviours (such as agent interaction protocols) the use of explicit state variables can be cumbersome. For this reason, JADE follows a compositional approach to allow application developers to build their own behaviours on the basis of the simpler ones directly provided by the framework. Applying the *Composite* design pattern [34], the *ComplexBehaviour* class is itself a *Behaviour*, but can have an arbitrary number of sub-behaviours or *children*. This class also defines the two methods *addSubBehaviour(Behaviour)* and *removeSubBehaviour(Behaviour)*, making it possible to define recursive aggregations of several sub-behaviours. Since *ComplexBehaviour* extends *Behaviour*, the agent writer has the possibility of implementing a structured tree composed of behaviours of different kinds (including *ComplexBehaviours* themselves). The agent scheduler only considers 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.

Besides the *ComplexBehaviour*, the JADE framework defines some other direct subclasses of *Behaviour*: the *SimpleBehaviour* class can be used by the agent developer to implement atomic steps of the agent work. A behaviour implemented by a subclass of *SimpleBehaviour* is executed by the JADE scheduler in a single time frame. Two more subclasses carry out specific actions: *SenderBehaviour* and *ReceiverBehaviour*. Notice that neither of these classes is abstract, so they can be directly instantiated passing appropriate parameters to their constructors. *SenderBehaviour* makes it possible to send a message, while *ReceiverBehaviour* makes it possible to receive a message which can be matched against a pattern. The behaviour blocks itself (without stopping all other agent activities) if no suitable messages are present in the queue.

The two classes *ComplexBehaviour* and *SimpleBehaviour* leave to their subclasses the choice of their termination condition and, for complex behaviours, the actual children scheduling policy. For atomic behaviours, two subclasses are provided. *OneShotBehaviour* is an abstract class that models behaviours that have to be executed only once. *CyclicBehaviour* is an abstract class that models behaviours that never end and have to be executed continuously. The *ComplexBehaviour* class fulfils the responsibility of organizing its children, but it defers children scheduling policy to subclasses. JADE provides ready made subclasses with common policies, but application programmers are free to implement their own.

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

NonDeterministicBehaviour is a *ComplexBehaviour* that executes its children behaviours non-deterministically, it blocks when all its children are blocked and it terminates when a certain condition on its sub-behaviours is met. The following conditions have been implemented: ending when all its sub-behaviours are performed, when any one among its sub-behaviours terminates or when at least N sub-behaviours have finished.

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

Copyright © 2001 John Wiley & Sons, Ltd.

Thinking in terms of patterns [34], 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 the top-level nodes of the behaviour aggregation tree, but every composite behaviour is responsible for its children's scheduling within its time frame. Likewise, when a behaviour object is blocked or restarted a notification mechanism built around a bi-directional *Chain of Responsibility* scheme provides all necessary event propagation.

Interaction protocols

Interaction protocols are used to design agents' interaction, providing a sequence of acceptable messages and a semantic for those messages. FIPA defines a set of general-purpose interaction protocols that provide a well-known interaction means to promote the social and cooperative nature of agents.

While each FIPA ACL message kind is given a formal semantics based on the speech-act theory, this is still not enough to satisfy the need for sociality of agent systems. This is because a typical agent interaction encompasses more than a single message, so more comprehensive abstractions are needed. FIPA specifications provide this abstraction as a collection of interaction protocols. When an agent engages in a conversation in accordance with some interaction protocol, it can play two roles: the *initiator* role is the one that creates the conversation, by sending the first ACL message of the new conversation; the *responder* role is the one that receives the first ACL message.

FIPA standard interaction protocols

The *fipa-request* protocol allows an agent to request another agent to perform some action; this is similar to ordinary request/response protocols used in client/server systems, but with a significant difference. Since software agents are autonomous entities, an agent can refuse to perform the requested action even if able to do so. So, while a client/server call either succeeds or fails raising an exception, a *fipa-request* interaction can succeed, can fail for a lack of the receiver-agent capabilities, but can also fail for the unwillingness of the receiver to perform the task at hand.

The *fipa-query* protocol is to be used to request an agent to give some information to the protocol initiator. Since queries to an agent knowledge are not supposed to interact with external software systems, the *fipa-query* protocol shares the outcome set with the *fipa-request* protocol, but has a flat structure.

The *fipa-request-when* protocol is a variation of the *fipa-request* protocol. The initiator agent requests the responder agent to perform an action in the future, and after an initiator-supplied precondition is valid. At that time the responder agent will try to execute the action and will notify the initiator using an inform message if the attempt is successful, or using a failure message.

The *fipa-contract-net* protocol is the first of the higher level FIPA interaction protocols. An agent, called the *manager*, can initiate a *fipa-contract-net* protocol sending a call for proposal, or a *cfp*, message to a set of other agents. Some agents will answer *not-understood* or *refuse* and will be discarded, but some others will hopefully send back a *propose* message, declaring their willingness to perform the requested action under a constraint set, such as price, time, etc. The manager agent can then evaluate all the proposals and select one from among them in accordance with some criterion.

Copyright © 2001 John Wiley & Sons, Ltd.

The *fipa-iterated-contract-net* protocol makes it possible for the manager to incrementally refine its call for proposals until a suitable contract is made. The only difference with respect to what was described previously is that a manager can refuse all proposals, with *reject-proposal* messages, and issue a revised *cfp* using the same conversation identifier.

fipa-auction-english protocol resembles an auction for selling goods, performed in accordance with the English style, using a low initial price and raising it gradually until no buyer declares his intention to pay. At that point, the last buyer can, and must, acquire the goods for sale, paying the amount he or she offered before.

The *fipa-auction-dutch* protocol follows the Dutch auctions style, which is used for the flowers market and works in the opposite manner to the English auction. The auctioneer starts at a price much higher than the real market value of the goods he or she is trying to sell, then lowers the price gradually until one of the buyers accepts the suggested price and buys the goods.

Implementing interaction protocols with JADE

JADE implements the communication policies between agents by means of protocols defining the set of possible conversations between the agents. In particular, for every protocol that JADE supports a couple of classes is provided: one for agents playing the initiator role and another for responder agents. For each designed protocol, a semantic is also provided so that third-parties agents can interact with the implemented ones. In addition to providing FIPA interaction protocols, JADE allows the definition of new interaction protocols. However, the best idea is probably to create interaction protocols as a composition of FIPA protocols in order to ease their implementation with JADE and to support interoperability with other FIPA-compliant systems.

To show how JADE supports FIPA interaction protocols, the *FipaRequestInitiatorBehaviour* class will be used as an example and this will also give a practical application of the JADE concurrency model. This class must be used by agents willing to start new *fipa-request* conversations, asking some other agent to do something for them.

Considering the FIPA97 specification, one finds a graphic representation of the protocol reported in Figure 5. The white boxes in the figure represent communicative acts that have to be issued by the initiator agent, whereas the grey ones represent communicative acts issued by the receiver agent. So, from the initiator point of view, ACL messages corresponding to white boxes have to be sent and the ones corresponding to grey boxes are to be received. From the figure it can also be seen that, after sending a *request* message, the initiator can expect either a *not-understood*, a *refuse* or an *agree* message. In the last case, another message will arrive, i.e. either a *failure* (some problem occurred), or an *inform* (in two different flavours; for actions with or without a result).

Looking at the documentation of the *FipaRequestInitiatorBehaviour* class, one finds a constructor accepting as a parameter the ACL *request* message that will start the conversation. Various abstract methods are provided to handle the various kinds of messages that can arrive during the protocol. Application programmers simply write their subclasses, implementing the abstract methods, named *handleAgree()*, *handleRefuse()*, and so on. This approach relies on classic OO techniques such as abstract classes and tries to present a familiar programming style to application developers, but the inner workings of *FipaRequestInitiatorBehaviour* are different.

Copyright © 2001 John Wiley & Sons, Ltd.

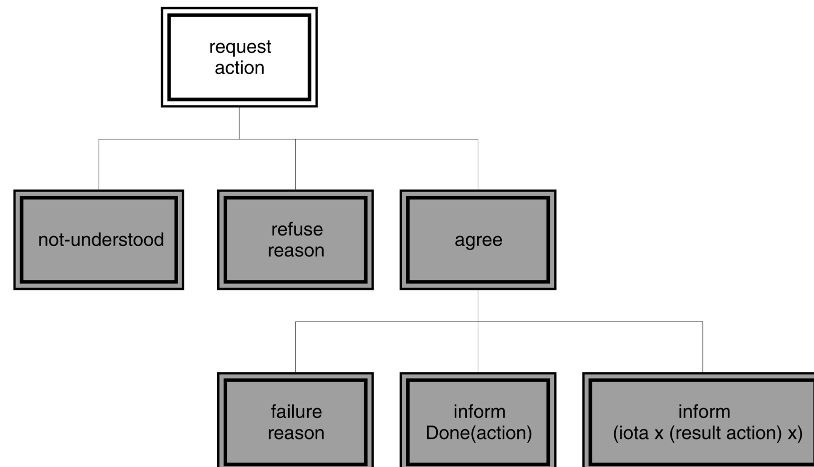

Figure 5. FIPA request interaction protocol.

This class relies on several private inner classes, which build an interaction protocol engine based on delegation, fully exploiting the JADE intra-agent concurrency model. Looking again at Figure 5, the following algorithm can be written:

Send the request message.
Receive the first reply and handle it.
If the first reply was an agreement, then receive the second reply and handle it.

When a complex behaviour comprises a list of steps, the JADE library offers the *SequentialBehaviour* class, of which *FipaRequestInitiatorBehaviour* will be a subclass. A *SenderBehaviour* can easily be used to carry out the first step, while the second is more interesting.

The second step is a disjunction of three possibilities, seen as branches in the graph shown in Figure 5. It is completed when any one of the three events (receiving a *not-understood*, receiving a *refuse* or receiving an *agree*) occurs. This observation makes it possible to exploit the *NonDeterministicBehaviour* class in order to express the three possible branches as concurrent activities. The non-deterministic behaviour object will be constructed so that it terminates as soon as one of its children receives something.

A similar approach is followed for the third step, but here the second *NonDeterministicBehaviour* object will be added to the main *FipaRequestInitiatorBehaviour* by the handler of the *agree* message. All the atomic behaviours are implemented as *SimpleBehaviour* subclasses, which first receive their specific kind of message (using JADE pattern matching capabilities) and then invoke the handler method (e.g. *handleNotUnderstood()*). The overall object structure is indicated in Figure 6, which also shows how similar the behaviour structure is to the original protocol structure in Figure 5.

Copyright © 2001 John Wiley & Sons, Ltd.

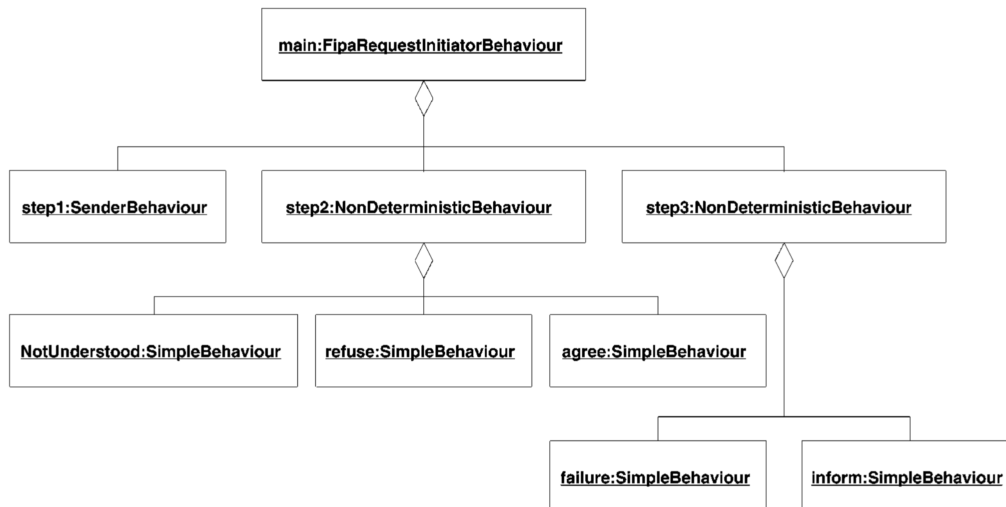

Figure 6. Internal structure of a JADE *FipaRequestInitiatorBehaviour* behaviour object.

DISCUSSION

The JADE design tries to bring together abstraction and efficiency, giving programmers easy access to the main FIPA standard assets while influencing run-time 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 choose the best transport available, to the address management module, that uses optimistic caching and direct connections between different containers.

The concurrency architecture attempts to be efficient not only in execution time but also in resource consumption. JADE behaviours invite the usage of cooperative scheduling for most situations, but do not prevent programmers from using the heavier Java multi-threading when some special need exists.

The following subsections discuss JADE under three important aspects. First, some performance oriented considerations are made; then, the robustness and reliability of JADE is taken into account, dealing with scalability and fault tolerance issues; finally, the existing applications using JADE are briefly introduced. The hope is that this will make it easier for the reader to assess the software system described in this paper.

JADE performance

Even though JADE was designed with efficiency in mind, obtaining 'high performance by design' is little more than wishful thinking, and real measurement must be performed to locate performance bottlenecks and to select the optimizations with the highest payoffs. During its first two years, JADE

Copyright © 2001 John Wiley & Sons, Ltd.

has been more concerned with standard compliance and with ease of use, so no complete performance measurements were taken: JADE was simply fast enough for the applications developed on it.

However, as part of a more general evaluation of JADE and other FIPA compliant agent platforms [37], some measurements were performed on JADE, comparing the standard IIOP transport with the local event dispatching mechanism (RMI was not taken into account, in order to have a fair comparison between JADE and other FIPA agent platforms, which are not distributed). The results showed that local message delivery was always more than 30 times faster than IIOP, with a 99% confidence level (i.e. at least 99 times out of 100). Moreover, local message delivery took about 10 ms, regardless of the overall load on the platform. This means that agent clusters located within the same agent container can exchange many messages without contributing much to the total workload on the messaging subsystem.

While these results are encouraging, more comprehensive performance measurements are needed of both JADE alone and of the applications based on JADE, in order to clearly assess JADE from a performance standpoint.

JADE robustness

Since JADE is a middle-ware 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 is involved in.

Scalability with respect to the number of agents is very limited in multi-agent systems, because it clashes with the autonomy requirement. In distributed object systems one can have many objects living within a single process, using resource pooling techniques to multiplex a much smaller number of threads and connections over all the objects. This resource overbooking approach works because, at a certain point in time, only a small subset of the objects will have pending calls and, in a reactive system, an object without external calls is idle. But agents are active objects, with an internal control flow that can start new communications at any time, so it is not possible to assume that an agent is idle just because it is not involved in any conversation.

Nevertheless, JADE tries to support large multi-agent systems as much 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 decentralized 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.

However, here the single message queue becomes a weakness of JADE agents, because it is a simple linear list which must be scanned in order to receive a message matching a certain template and this impairs scalability with respect to the number of messages. Future improvements could include hashing incoming messages to speed up pattern matching or even agents with many message queues.

Copyright © 2001 John Wiley & Sons, Ltd.

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; only a new behaviour object is created. So the only overhead associated with starting conversations is the behaviour object creation time and its memory occupation. Agents that are particularly sensitive to these overheads can easily bound them a priori, implementing a behaviour pool.

As can be seen from what is described in the Section '*Address management and caching*', the JADE Global Agent Descriptor Table (GADT) is centralized, and this raises problems with respect to both scalability and fault tolerance. Even though the JADE logical topology is a star, with a front end container acting as the hub, physical messages travel between any pair of containers with a single network hop, because every container can directly address any other (with RMI or IIOP). Moreover, the address caching techniques discussed above ensure that, as soon as agents have become known to each other, the central GADT is never looked up again. From a fault tolerance point of view, JADE does not perform very well because of the single point of failure represented by the front end container and, in particular, by the AMS and the default DF. Restarting the IIOP front end server is not a particularly complex issue (some technological problems still exist, though, since the CORBA ORB included in JDK 1.2 does not yet support persistent object references), and a complete solution can be organized around JADE optimistic address caching.

When an exception arises during an RMI communication with the front end container, a newer RMI object reference can be obtained by looking up the RMI registry again (which, in JADE, is bundled with the front end container and can be started on a user specified TCP port). The real problem is the AMS, described by FIPA97 specifications as a single agent; in the presence of a distributed agent platform, a replicated AMS would be necessary to grant complete fault tolerance of the platform. Nevertheless, it should be noted that, due to JADE decentralized 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.

JADE applications

Even if JADE is a young project and was designed with criteria more academic than industrial, and even if it has been released under an Open Source License only recently, it has already been used by a number of projects. In particular, it has been used in the three projects sponsored by the European Commission that are described below.

FACTS [38] is a project in the framework of the ACTS programme of the European Commission that 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 captured and refined over time through the collaboration of agents with different capabilities. The second application domain deals with agents in collaboration, and at the same time competing, in order to help the user to purchase a business trip. A Personal Travel Assistant represents the interests of the user and cooperates with a Travel Broker Agent in order to select and recommend the business trip.

CoMMA [39] 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 organization corporate memory and, in particular,

to facilitate the creation, dissemination, transmission and reuse of knowledge in the organization intranet.

JADE has also been used within the Living Memory (LiMe) project [40] to create a multi-agent system for the enhancement of social interaction within connected communities. LiMe is a Long Term Research Programme of the European Commission under its 'I-cubed' (Intelligent Information Interfaces) programme. JADE has successfully supported the LiMe communicating agents for dynamic user profiling, collective information dissemination and memory management for a 2-day field trial.

At the beginning of 1999, during a FIPA meeting in Seoul, JADE participated in the inter-operability tests with some other FIPA compliant platforms (ASL of Broadcom [16], MECCA of Siemens [41] and the agent platform of Comtec [42]). The results of the tests demonstrated that JADE is very near to offering full inter-operability with the other platforms passing a large part of the tests [43]. The failure on a few tests depended only on some incomplete definitions in the then current FIPA specifications that caused implementation mismatches among different platforms. The test results were very important because the comparison with other platforms allowed us to correct and improve JADE implementation.

CONCLUSIONS

In this paper we presented JADE (Java Agent Development framework), a software framework to aid the development of agent applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems. JADE is written in the Java language and comprises various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Java was the chosen programming language because of its many attractive features, which are particularly geared towards object-oriented programming in distributed heterogeneous environments. These features include Object Serialization, Reflection API and Remote Method Invocation (RMI). Starting from the FIPA assumption that only the external behaviour of system components should be specified, leaving the implementation details and internal architectures to agent developers, we produced a very general but primitive agent model that can serve as a useful basis to implement, for example, reactive or BDI architectures. In addition, the behaviour abstraction of our agent model permits an easy integration of external software. For example, we created a JessBehaviour that makes it possible to use an JESS as agent reasoning engine. In comparison to the agent development tools introduced in the previous section, JADE offers a more efficient implementation and a more general agent model. Such an agent model is more 'primitive' than the agent models offered, for example, by AgentBuilder and RETSINA; however, the overhead given by such sophisticated agent models might not be justified for agents that have to perform some simple tasks. In addition, sophisticated agent models such as BDI and reactive architectures, as previously mentioned, can be implemented on top of our 'primitive' agents model.

The development of JADE has not yet terminated. Our intention is, initially, to add the support for agent mobility and to provide some new tools to ease the development of agent systems such as, for example, a visual tool to compose agent behaviours, and to offer some higher level agent architecture as, for example, BDI architecture.

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

Copyright © 2001 John Wiley & Sons, Ltd.

REFERENCES

1. Maes P. Agents that reduce work and information overload. *Communications of the ACM* 1994; **37**(7):30–40.
2. Genesereth MR, Ketchpel SP. Software agents. *Communications of the ACM* 1994; **37**(7):48–53.
3. Wooldrige M, Jennings NR. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 1995; **10**(2):115–152.
4. Nwana HS. Software agents: An overview. *Knowledge Engineering Review* 1996; **11**(3):205–244.
5. Bradshaw JM. *Software Agents*. MIT Press: Cambridge, MA, 1997.
6. Jennings NR, Wooldrige M. *Agent Technology: Foundations, Applications, and Markets*. Springer: Berlin, 1998.
7. Rao AS, Georgeff MP. BDI agents: From theory to practice. *Proceedings of the First International Conference on Multi-Agent Systems*, San Francisco, CA, 1995; 312–319.
8. Sycara K, Pannu A, Williamson M, Zeng D. Distributed intelligent agents. *IEEE Expert* 1996; **11**(6):36–46.
9. Baumann J, Hohl F, Rothermel K, Straßer M. Mole—Concepts of a mobile agent system. *World Wide Web* 1998; **1**(3):123–137.
10. Patil RS, Fikes RE, Patel-Scheneider PF, McKay D, Finin T, Gruber T, Neches R. The DARPA knowledge sharing effort: Progress report. *Proceedings of the Third Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, 1992; 103–114.
11. Milojicic D, Breugst M, Busse I, Campbell J, Covaci S, Friedman B, Kosaka K, Lange D, Ono K, Oshima M, Tham C, Virdhagriswaran S, White J. MASIF—The OMG mobile agent system interoperability facility. *Proceedings of the 2nd International Workshop of Mobile Agents (MA '98)* (*Lecture Notes in Computer Science*, vol. 1477), Rothermel K, Hohl F (eds.). Springer: Stuttgart, 1998; 50–67.
12. Foundation for Intelligent Physical Agents. Specifications. http://www.fipa.org [1999].
13. Finin T, Labrou Y. KQML as an agent communication language. *Software Agents*, Bradshaw JM (eds.). MIT Press: Cambridge, MA, 1997; 291–316.
14. Reticular Systems. Agent Construction Tools. http://www.agentbuilder.com [1999].
15. Reticular Systems. AgentBuilder—An integrated toolkit for constructing intelligence software agents. http://www.agentbuilder.com [1999].
16. Kerr D, O'Sullivan D, Evans R, Richardson R, Somers F. Experiences using intelligent agent technologies as a unifying approach to network and service management. *Proceedings of IS&N 98*, Antwerp, Belgium, 1998.
17. Kawamura T, Yoshioka N, Hasegawa T, Ohsuga A, Honiden S. Bee-gent: Bonding and encapsulation enhancement agent framework for development of distributed systems. *Proceedings of the 6th Asia–Pacific Software Engineering Conference*, 1999.
18. The FIPA-OS home page. http://www.nortelnetworks.com/products/announcements/fipa/index.html.
19. The Grasshopper home page. http://www.ikv.de/products/grasshopper.
20. Martin DL, Cheyer AJ, Moran DB. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 1998; **13**:91–128.
21. Nwana HS, Ndumu DT, Lee LC. ZEUS: An advanced tool-kit for engineering distributed multi-agent systems. *Proceedings of PAAM98*, London, UK, 1998; 377–391.
22. Shoham Y. Agent-oriented programming. *Artificial Intelligence* 1993; **60**(1):51–92.
23. Thomas SR. The PLACA agent programming language. *Lecture Notes in Artificial Intelligence*, Wooldrige MJ, Jennings NR (eds.). Springer-Verlag: Berlin, 1994; 355–370.
24. The JADE Project home page. http://sharon.cselt.it/projects/jade.
25. Searle JR. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press: Cambridge, UK, 1969.
26. Meyer B. *Object Oriented Software Construction* (2nd edn). Prentice-Hall, 1997.
27. Lavender G, Schmidt D. Active object: An object behavioural pattern for concurrent programming. *Pattern Languages of Program Design*, Vlissides JM, Coplien JO, Kerth NL (eds.). Addison-Wesley: Reading, MA, 1996.
28. Singh MP. Write asynchronous, run synchronous. *IEEE Internet Computing* 1999; **3**(2):4–5.
29. Object Management Group. 95-11-03: Common Services. http://www.omg.org [1997].
30. Chiola G, Ciaccio G. Implementing a low cost, low latency parallel platform. *Parallel Computing* 1997; **22**:1703–1717.
31. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C. The virtual interface architecture. *IEEE Micro* 1998; **18**(2):58–64.
32. Karamcheti V, Plevyak J, Chien A. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing* 1996; **37**:21–40.
33. Johnson RE, Zweig JM. Delegation in C++. *Journal of Object Oriented Programming* 1991; **4**(7):31–34.
34. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
35. Lea D. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley: Reading, MA, 1997.
36. Friedman-Hill. Java expert system shell. http://herzberg.ca.sandia.gov/jess [1998].

Copyright © 2001 John Wiley & Sons, Ltd.

37. Laukkanen M. Evaluation of JADE 1.2. Evaluation of FIPA-compliant agent platforms. *Master's Thesis*. http://sharon.cselt.it/projects/jade/jadeMikko.ps.
38. The FACTS Project home page. http://www.labs.bt.com/profsoc/facts/.
39. The CoMMA Project home page. http://www.ii.atos-group.com/sophia/comma/HomePage.htm.
40. The LiMe Project home page. http://www.ee.ic.ac.uk/tour/ResearchSections/IntelligentCommunications/lime.html.
41. Lux AD, Steiner D. Understanding cooperation: An agent's perspective. *Proceedings of ICMAS'95*, San Francisco, USA, 1995.
42. Suguri H. COMTEC agent platform. http://www.fipa.org/glointe.htm [1998].
43. Foundation for Intelligent Physical Agents. FIPA interoperability experiment in Seoul. http://www.fipa.org [1999].

Copyright © 2001 John Wiley & Sons, Ltd.