The Journal of Systems and Software 83 (2010) 467–484

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A modern approach to multiagent development

D. Vallejo *, J. Albusac, J.A. Mateos, C. Glez-Morcillo, L. Jimenez

School of Computer Science, University of Castilla-La Mancha, Paseo de la Universidad, 4, 13071 Ciudad Real, Spain

ARTICLE INFO

Article history:
Received 28 January 2009
Received in revised form 18 September 2009
Accepted 18 September 2009
Available online 27 September 2009

Keywords:
Agent technology
Multiagent architecture
Distributed artificial intelligence

ABSTRACT

Multiagent systems (MAS) development frameworks aim at facilitating the development and administration of agent-based applications. Currently relevant tools, such as JADE, offer huge possibilities but they are generally linked to a specific technology (commonly Java). This fact may limit some application domains when deploying MAS, such as low efficiency or programming language restrictions. To contribute to the evolution of multiagent development tools and to overcome these constraints, we introduce a multiagent platform based on the FIPA standards and built on top of a modern object-oriented middleware. Experimental results prove the scalability and the short response-time of the proposal and justify the design and development of modern tools to contribute the multiagent technology.

1. Introduction

Agent-Oriented Programming (AOP) (Jennings, 2000) is defined as a software development paradigm that combines Artificial Intelligence and Distributed Systems. Basically, AOP conceives an application as a set of software components named agents (Wooldridge and Jennings, 1995), which are autonomous and proactive entities that are able to communicate with one another. The architectural model of agent-based systems is inherently distributed and is based on a peer-to-peer scheme, where each agent cooperates and communicates with others when it is needed.

Currently, multiagent systems (MAS) are being applied as a solution to a wide range of problems (Weiss, 1999), such as planning, scheduling systems, real-time control, robotics, and more industrial fields. This expansion in the use of MAS is also captured in software engineering models (Jennings, 2000) based on the use of autonomous agents to solve complex and distributed problems, the definition of methodologies (Wooldridge and Ciancarini, 2001), the use of languages (Bordini et al., 2006), or even the adoption of standards (Foundation for Intelligent Physical Agents, 2002a). In other words, AOP is used to deal with the design of problems in which other approaches are insufficient or incomplete.

This evolution can also be appreciated in MAS software development platforms (see Table 1). Thanks to these tools, agent-oriented system developers can focus their work on the multiagent application instead of dealing with administrative issues. For the time being, JADE (Java Agent DEvelopment framework) (Bellifemine et al., 2008) is probably the most widespread agent-oriented middleware. JADE can be defined as a modular and distributed framework that facilitates the development of agent-based applications. This agent framework implements the agents' life cycle and the management logic by providing administrative tools to deploy, monitor, and debug multiagent systems (Bellifemine et al., 2007).

Developers commonly use some existing framework in order to do the final deployment and not to spend too time in management and communication issues. The main advantages of this choice are an easy configuration and the administrative facilities derived from the agent middleware. Nevertheless, the main drawback is that the developer will be conditioned by the characteristics of the chosen framework.

Most current agent-oriented middlewares (see Table 1 in Section 2.2), although developed with the idea of providing general-purpose tools applicable to a wide range of domains, have several limitations that are directly related to their particular developments:

- Most of them are linked to a technology that limits their expansion to some platforms (due to hardware or resource consumption constraints).
- Most of them use technologies based on Java, which decreases the performance due to the Java Virtual Machine technology (see the experimental results in Section 4.2).

Within this context, several application domains, such as the systems that need short response-time, require solutions that overcome the previous limitations. To address these issues, the agent community needs to provide solutions that contribute to

* Corresponding author. Fax: +34 926 295 354.
E-mail address: David.Vallejo@uclm.es (D. Vallejo).
URL: http://personal.oreto.inf-cr.uclm.es/dvallejo (D. Vallejo).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.09.036

Table 1
Currently relevant agent-oriented software development platforms.

Name	Language	Standard	Free	Relevant paper
JADE	Java	FIPA	Yes	Bellifemine et al. (2008)
Jadex	Java	FIPA	Yes	Pokahr et al. (2005)
Cougaar	Java	No	Yes	Helsinger and Wright (2005)
Agent factory	Java	FIPA	Yes	Ross et al. (2004)
JACK	Java	No	No	Winikoff (2006)

the evolution of platforms for developing multiagent systems and solve the limitations of existing frameworks. Therefore, one of our goals is to use a modern object-oriented communication middleware (Henning, 2004) that allows us to extend the platform proposed in this work to different programming languages, operating systems, and hardware environments.

Another critical design topic to take into account in this evolution is the adoption of standards to promote the interoperability between MAS. In this context, the FIPA (Foundation for Intelligent Physical Agents) committee defines the most widespread proposal within the multiagent field. In this way, another of the central themes of our approach consists in adopting the set of FIPA specifications for the development of MAS. In fact, the last goal is to keep spreading the agent technology by providing an agent platform that facilitates the agent management and answers the challenge posed in (Ricci and Nguyen, 2007) regarding the deployment of MAS in real environments.

To face these challenges, this work introduces the design and development of an agent platform that follows the FIPA specifications and facilitates the development of agent-oriented applications. The main goal of our approach is to provide an alternative agent development framework that overcomes the previously discussed limitations. The design of this agent platform is heavily based on the use of well-known software engineering issues, such as design patterns and templates, and supported by ZeroC ICE (Henning, 2004), a modern object-oriented communication middleware that provides tools, APIs, and libraries to build object-oriented client–server distributed applications. ICE represents a modern alternative to develop distributed systems, which is based on CORBA's principles but with a lower complexity and a higher cohesion to make easy its use and learning (Henning, 2006).

The justification of using an object-oriented approach is due to two main reasons: (i) agents are actually an evolution of objects but with new characteristics such as autonomy and proactiveness (Wooldridge, 2001), and (ii) FIPA semantics shows object-oriented notions to describe concepts and ontologies (Foundation for Intelligent Physical Agents, 2004). This choice facilitates both the development of the agent platform itself and the development of agent-based applications on behalf of external developers.

In order to validate the agent platform, we have run two series of tests that measure its response-time and efficiency. The first one is related to an environment with multiple computers in which the agents send a variable number of messages with a determined size. Within this context, we evaluate the total traffic, the time spent in spamming messages, and the processing time spent in receiving messages depending on the number of messages sent and the message size. The second one refers to the deployment of agents by means of the named agent factories. The goal of this set of tests is evaluating the impact of hibernating and activating agents implemented in different programming languages by considering the agent creation time and the agent reactivation time. Finally, we briefly describe two scenarios where the agent platform proposed in this work has been used to successfully deploy multiagent applications.

The rest of the paper is organised as follows. Section 2 positions our work in the context of existing standards and multiagent platforms. Section 3 describes and discusses in depth the design and the development of the different components of the proposed platform. Section 4 presents the experiments carried out to validate the work and its application in two real problems. Finally, Section 5 concludes the paper and suggests future research lines.

2. Related work

In the last decade there has been an important number of software developments in different MAS related fields, ranging from declarative and imperative languages, integrated development environments, to agent platforms and frameworks (see Bordini et al., 2006 for a recent survey). Since this paper introduces a framework for the development of FIPA related MAS, we position our work in the context of existing standards and platforms.

2.1. MAS standards

2.1.1. Foundation for Intelligent Physical Agents (FIPA)

The Foundation for Intelligent Physical Agents (FIPA) is an IEEE committee aimed at promoting agent-based technology and interoperability between agent-based applications. FIPA specifications provide a set of standards for agents to interoperate at different levels. One of the more relevant documents is the FIPA Abstract Architecture Specification (Foundation for Intelligent Physical Agents, 2002a). This document and its derived specifications define the abstract architecture proposed by FIPA for the development of MAS (see Fig. 1). The main advantage of adopting the set of FIPA standards is the maturity obtained after more than 12 years of producing specifications for heterogeneous and interacting agents and agent-based systems.

The FIPA Agent Management Specification (Foundation for Intelligent Physical Agents, 2004) establishes the agent management model of the agent platform, including the basic FIPA management services, the management ontologies, and the message transport model. The first relevant service is the Directory Facilitator (DF), which is responsible for providing agents with a yellow-pages service. Next, the Agent Management System acts as the platform manager and maintains a directory with the valid agent identifiers (AIDs) of the agents registered with the platform. Besides providing the agents with a white pages service, the AMS also manages the agent life cycle. Finally, the Message Transport Service (MTS) provides support to send and receive Agent Communication Language (ACL) messages.

2.1.2. OMG's agent PSIG

The mission of the Agent Platform Special Interest Group (OMG) (PSIG) is to work with the OMG (Object Management Group) platform in order to stimulate the creation of agent specifications directly related to OMG technology (Odell, 2000). The relevant goals are to recommend OMG extensions to deal with agent tech-

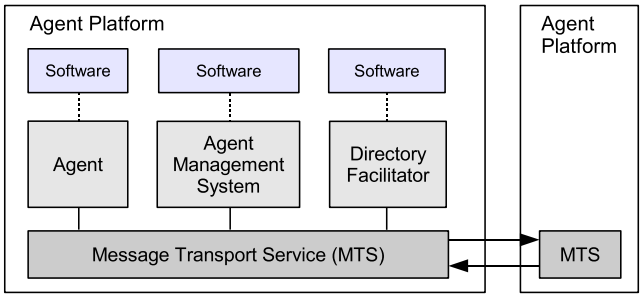

Fig. 1. FIPA abstract architecture.

nology, to promote standard modelling languages to increase consistency, and to help developers to understand how to develop MAS. The current status of the PSIG involves three areas: (i) notion of Agent in UPMS (UML Metamodel and Profile for Services), (ii) issue of a first Agent Metamodel and Profile Request for Proposal for notions required by agents, and (iii) inclusion of agent-based elements for the EML (Event Metamodel and Profile). However, this initiative is not as mature as FIPA is and it still needs to improve its specifications.

2.2. Relevant multiagent platforms

As previously mentioned, there is a high number of agent development frameworks that have been used to build MAS (Bordini et al., 2006). Within this context, JADE (Bellifemine et al., 2008) represents the most mature framework and, after 10 years of development, it is a reference for the development of FIPA-compliant MAS. However, there exist other relevant agent platforms that have contributed to the expansion of this kind of tools. The most relevant ones are summarised in Table 1.

Jadex is an open-source software framework to design goal-oriented agents that follow the BDI model (Pokahr et al., 2005). This is a relevant advantage since the BDI model has been widely used for years. The main goal of the project is to develop agent-based systems in an easy and intuitive way by providing a middleware using well-known engineering foundations. Jadex allows developers to build software agents in XML and Java and can be deployed on different middlewares, such as JADE. This tool has been applied in several projects, such as treatment scheduling for patients in hospitals, games, and teaching issues. The main weakness of Jadex is that it is limited to Java technology and this may involve an important restriction in some areas/applications.

Cougaar (Helsinger and Wright, 2005) is an open-source, Java-based architecture to develop distributed agent-based applications. One of the main objectives of this project is to provide a tool for large-scale and flexible MAS. However, it does not make use of standards so that interoperability with other systems is not achieved. Agent factory (Ross et al., 2004) can be understood as an extensible framework to develop and deploy agent-oriented applications and aims at giving support to the development of complex distributed systems in a wide range of application domains. The main characteristics of this framework are the adoption of FIPA standards, the use of a *plug and play* philosophy to deploy and configure multiple agent types, the use of Java, and the inclusion of some agent interpreters. A strength of this project is the inclusion of a lightweight version named J2ME MIDP compliant Agent Platform (AFME). However, both versions are completely based on Java, which may be a limitation for certain applications.

Another relevant work is JACK (Winikoff, 2006), a commercial environment for building, running, and deploying MAS by using a component-based approach. It also provides a Java-based language to support agent-oriented concepts such as agents, capabilities, events, or plans; and an editor for agents to build plans. Its main weakness is that it does not follow open standards and it only provides a commercial version.

2.2.1. Java Agent DEvelopment Framework (JADE)

JADE (Bellifemine et al., 2008) is probably the most widespread FIPA-compliant agent-oriented middleware. From a practical point of view, JADE is organised in two development packages: (i) a FIPA-compliant multiagent system and (ii) a package for developing Java-based agents. Therefore, developers making use of JADE must implement their agents in Java by following the developer guide provided by JADE (Bellifemine et al., 2007). In contrast to this approach, we suggest the use of a modern middleware to cover a higher number of programming languages and to provide the multiagent platform with native services that facilitate the development and management of MAS, such as transparent location service, implicit server activation, transparent replication service, load-balancing policies, event distribution service, object persistence, firewall transversal service, software distribution and patching service...

Next, we discuss the main JADE's characteristics and establish a comparison between this platform and our approach. In JADE, the communication is carried out through Java RMI (Downing, 1998) so that the programming language used is Java. In contrast, we choose a modern middleware that allows to make use of a wide range of programming languages. In fact, we provide developers with the management agent implemented in C++, Java, and Python. Besides, we have paid special attention to the scalability of the system when deploying a high number of agents or exchanging a high number of messages. JADE agents are implemented as threads and are hosted in agent containers, which provide agents with the runtime environment. We use a similar approach but our agents are multi-threaded processes that attend requests concurrently instead of individual threads. JADE provides an administrative graphical user interface to manage the agent platform. In contrast, we have developed a web system for developers to deal with management issues via a web browser. Finally, JADE gives support to mobility for both code and agent state, offers automatic FIPA services deployment, and provides a FIPA interaction protocol library. In our platform, the FIPA services are automatically deployed as well and the agent platform provides transparent replication for these basic services. In this way, developers can build robust applications and distribute the system load depending on the application requirements. To deal with FIPA interaction protocols, we adopt an approach based on the use of callback objects as described in Section 3.2.

3. The multiagent platform

The platform presented in this work offers a new and modern alternative for the development and deployment of MAS. Our goal is to provide developers with a solution which solves some limitations of current tools and to keep the evolution of these tools on track. On the other hand, it also makes the development of agent-based solutions easier and provides interoperability by means of the use of FIPA specifications. Next, we summarise the most important characteristics of our proposal:

- Implementation of the basic management agent in various programming languages (C++, Java, and Python but extensible to C#, Visual Basic, and Ruby) for the user to choose the most suitable option by taking into account the application domain. This is possible thanks to the middleware that gives support to the agent platform, which allows us to provide implementations of the same component of the agent platform in different programming languages.
- Scalability to a high number of agents and short response-times (see experimental results discussed in Section 4.2.2).
- Use of a modern, efficient, and object-oriented communication middleware to guarantee interoperability among different programming languages, operating systems, hardware platforms, and communication networks.
- Platform based on the FIPA specifications for the development of MAS.
- Automatic deployment of FIPA basic management services: Agent Management System, Directory Facilitator, and Message Transport Service.
- Design based on well-known solutions in software engineering such as design patterns and templates to instantiate services and servers within the agent platform.

- Transparent location service to all elements of the agent platform.
- Robustness thanks to the automatic replication mechanisms provided.
- Authentication, encryption, and message integrity by means of the SSL protocol.
- Administrative GUI to facilitate the agent platform administration from anywhere with connection via a web browser.

3.1. Architectural overview

The architecture of the agent platform consists of a registry and a number of nodes. From an administrative point of view, there exists a coordination between the registry and the nodes to manage the information and the processes that compose the distributed application. Basically, each application allocates servers to nodes. The registry maintains a persistent record of the information linked to the application, while the nodes activate and monitor their associated servers. In this way, the relevant FIPA services and the agents are deployed in the nodes of the application (customised by the developer). Every node runs on a physical device that provides the resources needed to host servers, although more than one node can be run on a single device. Similarly, the developer is free to decide what devices to use when deploying the multiagent application.

In addition to the services specified by FIPA and the management agent, the agent factory is included to make the agent deployment in the nodes. This element implements the abstract factory pattern (Gamma et al., 1995) and allows to easily instantiate agents. On the other hand, the administrative interface directly interacts with the agent platform to perform the multiagent system operations. Through this interface, the developer can create agents, consult their state, or even destroy them. Besides, the developer is also provided with an administrative web-based system. The whole architecture is graphically shown in Fig. 2.

Before describing in depth the components of the agent platform, the general characteristics of the architectural design are summarised:

- Object location is transparent, that is, the platform maintains a complete independence between distributed objects, agents in our domain, and their physical location, that is, the hardware that hosts the servants. This point is relevant when carrying out the agent management and communication.
- Server replication is practically immediate (replication with state requires additional code). Thus, the services of the agent platform can be replicated to provide developers with a robust platform. Furthermore, the registry that stores information about nodes, servers, and objects can also be replicated to increase fault-tolerance. Finally, it is also possible to use load balancing between replicas of the same server in order to improve adaptability.
- Server activation can be on-demand so that the agent platform is able to efficiently manage the resources used.
- Easy integration with existing firewalls to deploy agents under hostile network environments.
- Easy and scalable application deployment thanks to the use of descriptors. These XML files specify the components of the distributed application and where the servers will be deployed.

3.2. Agent

The agent represents the basic management entity of the agent platform and, according to FIPA (Foundation for Intelligent Physical Agents, 2002a), is the computation process that implements the functions and communication mechanisms needed to meet the requirements of a concrete task.

In order to describe the design of our management agent, the classical middleware nomenclature has been taken into account. In this way, the management agent is represented and managed by a proxy, that is, the ambassador of the (possibly) remote object.

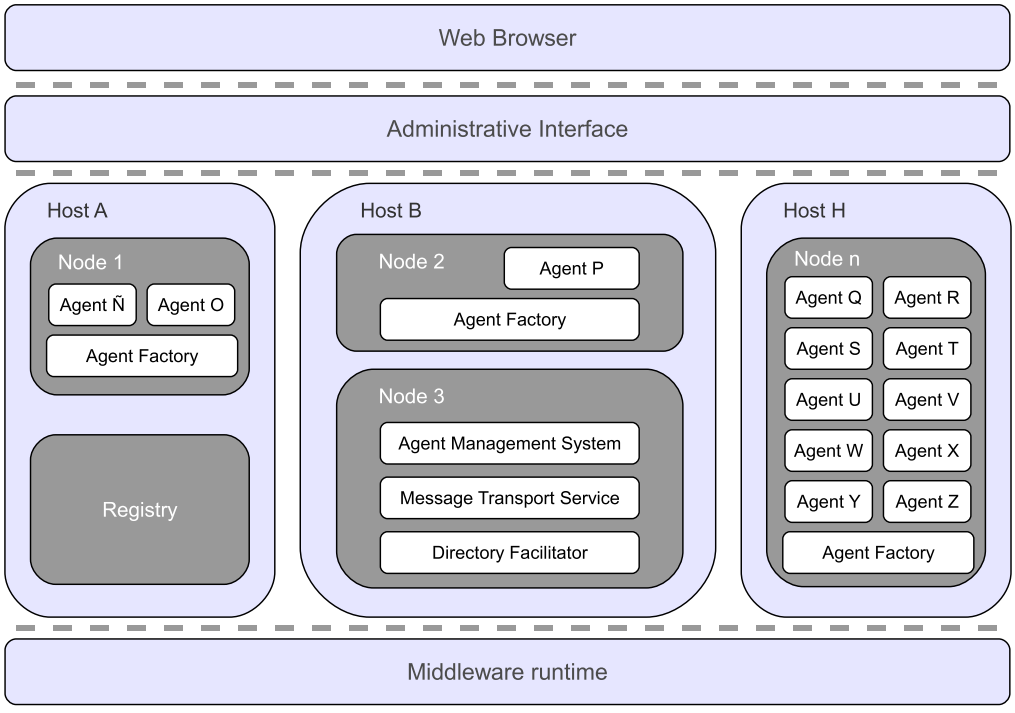

Fig. 2. General architecture of the agent platform.

Table 2
Relation between FIPA agent attributes and the ICE's proxy.

FIPA Agent	Proxy
Name	Object identity
Addresses	Proxy's endpoints
Resolvers	Location service registry's endpoints

So, requests made to the proxy will be translated into remote invocations to the physical implementation of the agent in a concrete programming language. Table 2 shows the relation between the semantics specified by FIPA in the *fipa-agent-management* ontology (Foundation for Intelligent Physical Agents, 2004) to define the agent and the information encapsulated in a proxy.

On the one hand, the proxy encapsulates the unique identifier of the agent, that is, the unique identifier of the distributed object. On the other hand, the proxy also specifies the physical addresses used to contact the server that supports the agent functionality. The last relevant parameter of the agent is *resolvers*, optionally specified by FIPA for the agent to be contacted. Our proposal does not need to introduce any type of resolver element because the middleware employed provides a location service (IceGrid). So far, our design remains as simple as possible.

To support the agent life cycle (Foundation for Intelligent Physical Agents, 2004), we adopt a state machine-based approach by taking into account the transitions between states specified by FIPA. The main issue is to guarantee a correct concurrent management of the agent state. For instance, the AMS may change the agent state while another element is simultaneously getting it. To do that we employ a monitor like mutual exclusion mechanism.

The third relevant characteristic of the agent design is persistence. In this way, the agent maintains its state, that is, it is possible to resume its execution and to recover it when activating the agent again. Additional state can be specified by the developer depending on the application requirements by means of metadata and minimal source code.

The agent interface is shown in Fig. 3, in which the functionality that offers to the rest of the platform members is described. However, this fact does not mean that the agent is submissive. On the contrary, it can decide what to do in view of external requests depending on its internal state. The agent operations are as follows:

- *setState*: allows to change the agent's state depending on its current state.
- *getState*: allows to get the current agent's state.
- *receiveMessage*: allows the agent to receive messages from other agents or Message Transport Systems.
- *dfAdv*: allows the agent to receive information from the yellow-pages services of the agent platform (Directory Facilitators). A previous subscription is required.
- *destroy*: ends the agent execution. The AMS is responsible for destroying agents when required.

Currently, the agent communication model is based on the use of the remote procedure call protocol provided by ICE, which can use TCP/IP or UDP as transport protocol. Furthermore, it is also possible to employ SSL for secure connections. The developer is responsible for making this choice depending on the application requirements. The main advantages of this low-level communication layer are

MTS

```
interface Comm {
 ["ami"] void receiveMessage
 (Message m);
};

interface MTS extends Comm {};
```

AMS

```
interface AMS extends Comm {
 void register (Agent* aid)
  throws AgentExists;
 idempotent void deregister
  (Ice::Identity id);
 idempotent Agent* search
  (Ice::Identity id);
 idempotent ApDescription
  getDescription ();
};
```

Agent Factory

```
interface AgentFactory {
 Agent* create
 (string name)
  throws AgentExists;
 AgentSeq createSeq
 (Ice::StringSeq names,
  out Ice::StringSeq registered);
};
```

Agent

```
interface Agent extends Comm {
 void setState
  (AgentState st);
 idempotent AgentState
  getState ();
 void dfAdv (DFAction act,
  DfAgentDescription desc);
 void destroy ();
};
```

DF

```
interface DF extends Comm{
 void register
 (DfAgentDescription desc)
  throws AgentExists;
 idempotent void deregister
  (Ice::Identity id);
 void modify
  (DfAgentDescription desc);
 idempotent AgentSeq search
  (DfAgentDescription desc);
 void subscribe (Agent* aid)
  throws AlreadySubscribed;
 idempotent void unsubscribe
  (Agent* aid);
};
```

Fig. 3. Public interfaces of the agent platform defined with Slice (ICE's IDL).

robustness and efficiency (ZeroC ICE Performance Analysis). On top of this layer we develop the FIPA interaction protocols for the agents to communicate. On the other hand, the main drawback of this choice is that the communication protocol is not standardised and it is not human readable as other protocols such as SOAP.

After having introduced the communication model, we will describe how the agents communicate with one another, that is, how they send and receive messages. Currently, the agent can receive messages synchronously and asynchronously (specified by the ["*ami*"] (asynchronous method invocation) metadata of interface *Comm* in Fig. 3). By means of this last way, the sender is not blocked until the receiver agent completely processes the remote invocation. In fact, the sender becomes free when the content of the message is transmitted. But, how does the sender know if the receiver correctly processed the message and carried out the requested actions? Our approach is based on callback objects, which are used to notify if there was some problem or the message was adequately received and processed. This idea allows us to easily model the FIPA interaction protocols, which usually involve messages to confirm or agree previously sent proposals or requests (see the FIPA Request Interaction Protocol (Foundation for Intelligent Physical Agents, 2002c) and Fig. 4).

The parameter accepted by the operation *receiveMessage* is a structure *Message* according to the *fipa-agent-management* ontology (Foundation for Intelligent Physical Agents, 2004). The most relevant design solution adopted consists in using the proxies to refer both the message sender and receivers. In this way, the agents' physical addresses are transparent for the message (in fact for all communication processes).

3.3. Agent factory

The agent factory represents the component of the agent platform which is responsible for creating agents and can be understood as the servant manager of the agents, that is, like the manager of the physical implementation of the agents in a particular programming language. Although this component is not explicitly defined by FIPA, it is interesting to perform an efficient management of the resources associated to the agents. The main advantages obtained through this component are the scalability when deploying the agents and the abstraction mechanism provided to deal with them.

From a design point of view, the agent factory is a container of agents' servants based on the abstract factory pattern (Gamma et al., 1995). Through this component, a developer is able to instantiate agents within the platform without worrying about management issues. Such management is focused on efficiently dealing with the resources needed to host the agent physical implementations. For instance, it is not practical that a system composed of thousands of agents allocates them all in RAM. In fact, providing a method that allows to activate agents on demand is more efficient, that is, it makes possible the use of resources only when needed.

The agent factory of this platform has been designed with that goal in mind. To do that, it only hosts a number of agents in memory (customised depending on the application requirements) and offers a mechanism for activating agents on-demand when required. Besides providing the agent platform with an efficient resource management, the system is scalable when the number of agents increases. Currently, the algorithm used to replace agents in memory is based on the *least recently used* approach, although the factory can be extended with other techniques.

An important question to discuss is why the agent factories do not provide operations to destroy agents. If the agent factory allows to create agents, it makes sense that this component allows to destroy them. However, this is not a good design choice for several reasons. The main one is that a certain platform component (such as the AMS) does not only need to have a reference to the agent but also to the agent factory that created it. In other words, for one deployed agent every component interested in interacting with such agent must have deal with two references: one to the agent and another to the agent factory that previously instantiated it. If this platform is used to deploy a high number of agents and agent factories, then an alternative design is needed. This is why the operation *destroy* belongs to the agent itself instead of the agent factory. Furthermore, this approach allows the agent to decide whether a external entity can destroy it.

The interface of the agent factory is shown in Fig. 3 and is composed of the following operations:

- *create*: allows to create an agent within the agent platform.
- *createSeq*: allows to create a set of agents within the agent platform.
- *destroy*: ends the agent factory execution and free all the related resources.

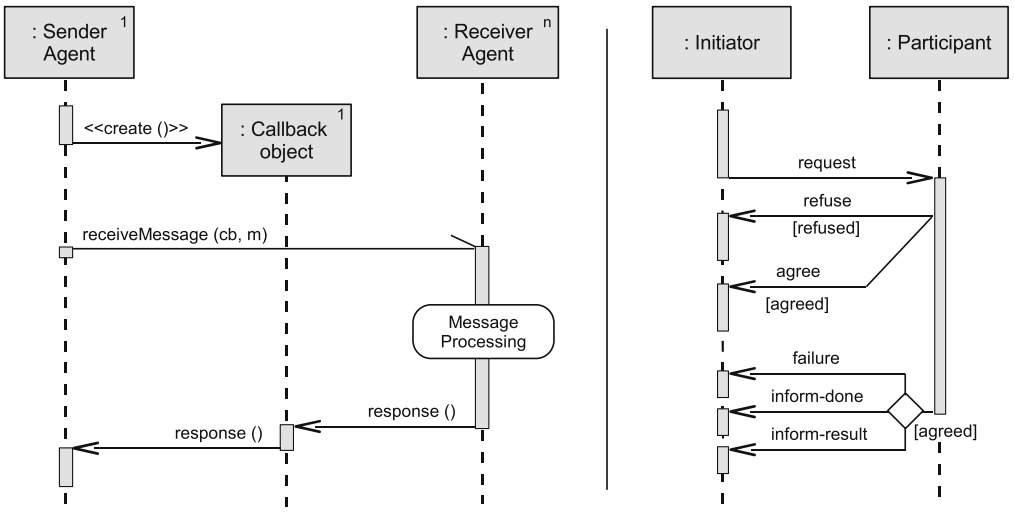

Fig. 4. FIPA Request Interaction Protocol modelled with a callback object.

The communication with the agent factory is done via the previously discussed public interface by means of remote invocations. The platform allows the interaction with the agent factories through the administrative web interface. Besides, the developer can also interact with the agent factories programmatically.

3.4. Agent Management System (AMS)

The Agent Management System (AMS) is the FIPA component that manages the control and access to the agent platform (Foundation for Intelligent Physical Agents, 2004). Only one AMS is allowed in an agent platform, which is responsible for maintaining a directory with all the agents registered with the platform. In other words, the AMS provides a white-pages service to the other components of the system. On the other hand, it is the main management element because all the agents must register with it in order to obtain a valid identifier within the agent platform.

Since the AMS is the core of the agent platform, the design of this component is driven by robustness and fault-tolerance. To achieve this general goal, our design is based on associating the AMS with a well-known object that can be replicated to avoid possible failures in this critical component of the system. In order for the agent to register with the agent platform, it has to obtain a proxy to the AMS and invoke the operation *register*. Internally, the AMS maintains a list with all the registered agents. Therefore, a problem arises when replicating this service in different nodes (in the same or different machines) as the AMS's state (the agent directory) must be consistent for all the replicas.

To overcome this problem, we adopt a master–slave replication mechanism. One of the AMS replicas is considered like the master and the rest like the slaves, and every replica makes use of the observer design pattern (Gamma et al., 1995) to establish a connection with the master. Basically, each one of the slaves observes the master state: when the master updates its state (the agent directory), it sends notifications to the slaves. In the same way, every slave maintains an internal directory that can only be modified through the observer's notifications (see Fig. 5). If the master is not able to send a notification to a slave, then the session previously established between them will be destroyed and the slave will have to establish a new session and synchronise its state with the master. Thanks to the use of sessions, the correct synchronisation between the master and the slaves is guaranteed.

Another question to take into account is the slave promotion when the master goes down. We propose two possible solutions: using a coordination mechanism among the slaves to randomly choose a master or using a priority-based algorithm. In the current version of the platform, we customise the replicas by means of a numeric priority (lower values imply a higher priority) so that the slave with the highest priority will be promoted to master when it is needed. The slaves communicate among themselves to find out the one with the highest priority.

The interface of the AMS is shown in Fig. 3 and involves operations for registering and searching agents:

- *register*: allows to register an agent within the agent platform. The AMS gets the agent-identifier from the object's identity associated to the proxy parameter.
- *deregister*: allows to deregister an agent. The security mechanism used consists in checking if the component that invoked the operation has the same identity that the proxy related identity, because the proxy represents the agent servant.
- *search*: represents the white-pages service provided by the AMS.
- *get-description*: allows to obtain the services provided within the agent platform. In fact, this description lets external agents to contact to the agents for giving support to the desired service.
- *receiveMessage*: allows the AMS to receive messages from other agents.

Before internally registering with the agent platform, an agent needs to obtain a proxy to the AMS. To do that, there are two possible options:

1. To assume that the AMS identifier is *AMS@platform_name*, create the proxy, check that the object exists, and register with it.
2. To consult the registry of the middleware itself asking by name (*AMS*) or by service (*::FIPA::AMS*).

It is important to remark that the developer can choose between using system interfaces or the suitable FIPA Interaction Protocol for agents to interact with the FIPA basic services and between themselves. In this work, we internally assume a direct

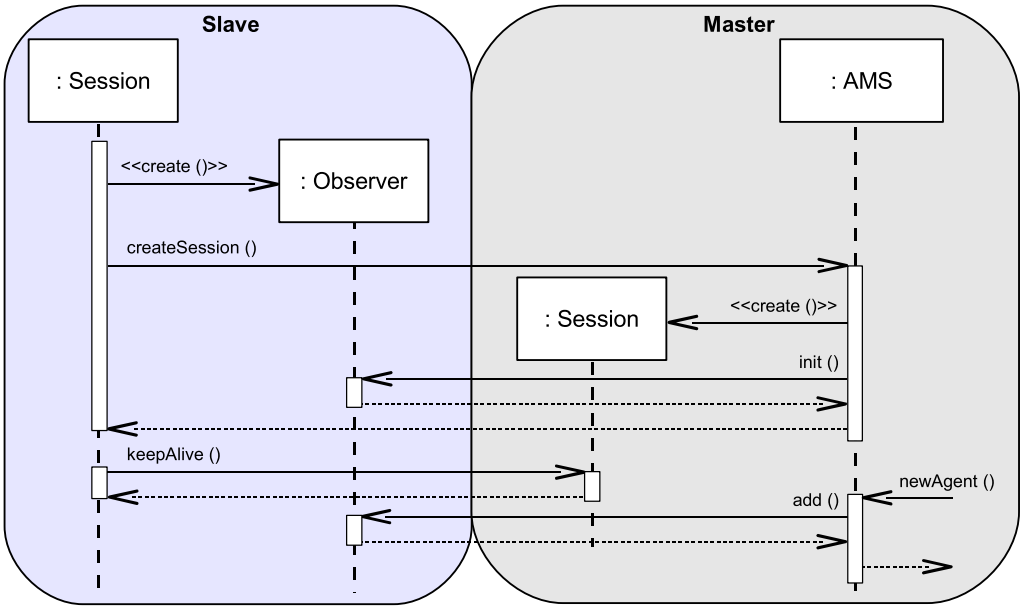

Fig. 5. Master–slave replication mechanism adopted by both the AMS and DF.

interaction by means of the interfaces for efficiency reasons. On the other hand, the communication among AMS replicas is transparent to both the agents and the developer of the multiagent application.

3.5. Directory Facilitator (DF)

FIPA defines the Directory Facilitator (DF) (Foundation for Intelligent Physical Agents, 2004) as an optional component of the agent platform that provides yellow-pages service. The agents must register with the DF in order to be able to interact with it. FIPA specifies that more than one DF instances can exist in the same agent platform, optionally organised into a hierarchy.

Although the inclusion of this component is not mandatory according to FIPA, its inclusion is very interesting in the first version of this platform if we take into account the increasing popularity of service-oriented computing (Huhns and Singh, 2005). The DF maintains an internal directory similar to AMS's so the design solutions applied to the AMS have been used with the DF as well (concurrency control and master–slave replication). However, there exists one particular function of the DF that requires special attention: the service notification mechanism. This functionality allows previously subscribed agents to get information of the services provided by other agents. For instance, if an agent registers with the DF, the rest of subscribed agents will be informed about the recently subscribed agent and the services that it offers.

On the other hand, and with the goal of extending the DF's functions, this service has been complemented with another service discovery protocol (Villa et al., 2007). This design decision makes possible the interaction with a high number of elements, being agents or not, that provide services within a particular context. In fact, the work presented in (Villa et al., 2007) was initially conceived for wireless sensor networks although extensible to any domain. Basically, this service discovery framework provides a simple mechanism to carry out service announcements by minimising configuration issues and maximising scalability. By means of this integration, the platform service management offers a scalable solution extensible to a wide range of domains.

The DF interface is similar to the AMS regarding management issues and is shown in Fig. 3. Main differences are as follows:

- *search*: allows the agents to search for other agents providing a concrete service.
- *subscribe*: used by the agents to subscribe to DF notifications.
- *unsubscribe*: used by the agents to unsubscribe to DF notifications.
- *receiveMessage*: allows the DF to receive messages from other agents.

Finally, the communication model is analogous to the AMS's.

3.6. Message Transport Service (MTS)

The Message Transport Service (MTS) gives support to the exchange of messages between agents in the same or different platforms (Foundation for Intelligent Physical Agents, 2002b). Since the messages are structured in the envelope and the payload according to FIPA, the MTS is responsible for analysing the envelope to send messages to the final destination.

The design of the MTS consists in a well-known object identified as *MTS@platform_name* within the agent platform. Since this service may be a bottleneck if agents exchange a high number of messages, a replication scheme has been adopted again. In this way, the developer of the multiagent application can replicate the MTS as many times as needed depending on the application requirements. For instance, the MTS may be deployed in *n* different computers to support the system load when the agents send messages. This component is stateless because it only receives, processes, and sends messages, so there is no need to use a synchronisation mechanism between the replicas.

The interface of the MTS is composed of a simple operation to receive messages (see Fig. 3). An agent asynchronously invokes this operation to send messages so that it becomes free when the MTS obtains the message and without needing to wait for the receivers to get the message.

To send messages, the MTS obtains the information needed from the message envelope, that is, the receivers' addresses (or proxies in our platform).

3.7. Implementation details

The agent deployment in the proposed platform is done by means of the agent factories (called "agent containers" in JADE's terminology (Bellifemine et al., 2007)). These elements are built with the abstract factory pattern (Gamma et al., 1995) and used to manage the life cycle of the servants that give support to the agents in the different agent factories. In other words, a servant is the physical implementation of an agent in a certain programming language. Therefore, every agent in the agent platform has a relation with only one agent factory. As previously mentioned, the basic management agent is implemented in three different programming languages: C++, Java, and Python. Nevertheless, it is possible to expand this implementation to the rest of languages supported by the middleware ZeroC ICE in order to cover a wider application range.

The agent factory makes use of Freeze, the ICE's persistence service. Concretely, its implementation is based on Freeze Evictor, which is employed to activate, manage, and deactivate agents. To do that, the developer has to explicitly specify what data are persistent. In the current implementation, agents' persistent data is the agent state defined by FIPA. To specify the operations that affect persistent data, we use metadata directives defined by ICE. In this way, the agent factory knows how to transparently manage agent persistence regarding the multiagent application. Currently, the agent factory with persistence is implemented in C++ and Java, respectively.

The AMS is contained within a server that implements the service configurator pattern (Jain and Schmidt, 1997). In this way, services are developed as components that can be dynamically loaded in general-purpose and highly configurable servers. Our solution is based on the deployment of a server to host the management services specified by FIPA, that is, the Agent Management System, the Directory Facilitator, and the Message Transport Service. The advantages of this approach are as follows:

- Interactions between the services available in the same server can be optimised.
- No need of compiling or linking when configuring a server.
- Services implemented within this server provide a common development framework and are easy to administrate.

To deal with this issue we have used IceBox, which is the ICE service that allows to dynamically load and unload servers. In the current version, the AMS is implemented in C++ in order to provide an efficient solution when dealing with the management operations within the agent platform. Therefore, the developers do not need to worry about how this component is implemented. In fact, they only need to know the AMS public interface to interact with this service. Finally, the operations *register*, *deregister*, and *search* are interlocked in order to guarantee the correct access to the agent directory. If we do not provide this mechanism, one agent may modify the directory while another agent is reading it.

The DF is hosted by an IceBox server and implemented in C++ in order to provide the agent platform with an efficient yellow-pages

service. To implement the DF, we have adopted a publish-subscribe mechanism so that the agents may be publishers of information, subscribers, or both. Within the context of the DF, this information consists in the service descriptions of the agents that compose the multiagent system. So, the communication model is driven by content instead of senders or receivers. In order to deal with this topic, we introduce the event channels as the communication elements used to send information.

On the other hand, the DFs federation is also addressed by FIPA. We tackle this issue by means of links between event channels. In this way, an event received by a specific DF may be forwarded to another DF depending on a certain criterion. Currently, we use the concept of *cost* to quantify both messages and links. The cost is used to filter messages, that is, a DF will forward a message if its cost is lower or equals than the cost of the link. To model broadcast messages, a DF with a link which cost is equals to 0 will forward all messages received and messages with cost equals to 0 will be forwarded on all links.

To implement this scheme we employ IceStorm, the publish-subscribe mechanism offered by ICE to deal with service notifications. In short, IceStorm gives support to event channels by decoupling the producers of information and the consumers, that is, the publishers and the subscribers. The main concepts are the message, related to the information to send, in our case service descriptions and agent subscriptions; and the topic, which represents the channel between publishers and subscribers.

Finally, the MTS implementation follows the same scheme than the other FIPA basic services, that is, it is based on the use of IceBox and is implemented in C++. In Section 4.2.1, we evaluate the response-time of this component in different scenarios.

3.8. System deployment

The multiagent platform is deployed through the following operations:

1. To create the directories needed to store persistent data.
2. To start a node that is used to host the FIPA basic management services: AMS, DF, and MTS. Initially, these services are inactive until they receive some request, that is, they activate on-demand. This node also hosts the application registry internally used to manage the location service.
3. To start a node that is used to host three agent factories, each one of them with the goal of creating agents implemented in three different programming languages: C++, Java, and Python. These factories are also activated on-demand.
4. To deploy the application descriptor, that is, a XML file that specifies what services will be activated, what are their properties, and what nodes will be used to allocate them (see Fig. 6).

This set of steps represents the default functionality of the platform presented in this work. However, it is possible to completely customise the multiagent system deployment depending on the user application:

- Number of hardware devices.
- Number of nodes (in the same or different hosts).
- Servers to deploy in each node. Note that the developer can replicate the FIPA basic services in different nodes and, therefore, in different hosts.

```
<icegrid>
   <application name="FIPA">
   ...
      <service-template id="MTSServiceTemplate">
         <parameter name="id" default="MTSService.${server}"/>
         <service name="${id}" entry="MTSServer:createMTSServer">
            <properties>
               <property name="adapter" value="${id}.Adapter"/>
            </properties>
            <adapter name="${id}.Adapter" endpoints="default"
              id="${server}.${service}.${id}.Adapter"
              replica-group="MTSServiceReplicaGroup"/>
      ...
      <server-template id="IceBoxMTSServerTemplate">
         <parameter name="index" default="0"/>
         <parameter name="instance-name" default="${application}.IceBoxMTSServer"/>
         <icebox id="${instance-name}.${index}" activation="on-demand" exe="icebox">
            <properties>
               <property name="IceBox.InstanceName" value="${server}"/>
               <property name="IceBox.ServiceManager.Endpoints"
                         value="tcp -h 127.0.0.1"/>
               <property name="Ice.ThreadPool.Server.Size" value="8"/>
               <property name="Ice.ThreadPool.Client.Size" value="8"/>
            </properties>
            <log path="../logs/IceBoxMTSServer.${index}.err" property="Ice.StdErr"/>
            <log path="../logs/IceBoxMTSServer.${index}.out" property="Ice.StdOut"/>
            <service-instance template="MTSServiceTemplate"/>
         </icebox>
      </server-template>
      ...
      <node name="FIPAServices">
         <description>The node that hosts the FIPA basic services.</description>
         <server-instance template="IceBoxFIPAServerTemplate"/>
         <server-instance template="IceBox"/>
      </node>
      <node name="Node1">
         <server-instance template="IceBoxAgentFactoryTemplate" index="0"/>
         <server-instance template="IceBoxMTSServerTemplate"/>
         <server-instance template="SnifferServerTemplate"/>
         <server-instance template="SpammerFactoryServerTemplate" index="0"/>
      </node>
      ...
   </application>
</icegrid>
```

Fig. 6. Some relevant pieces of the application descriptor.

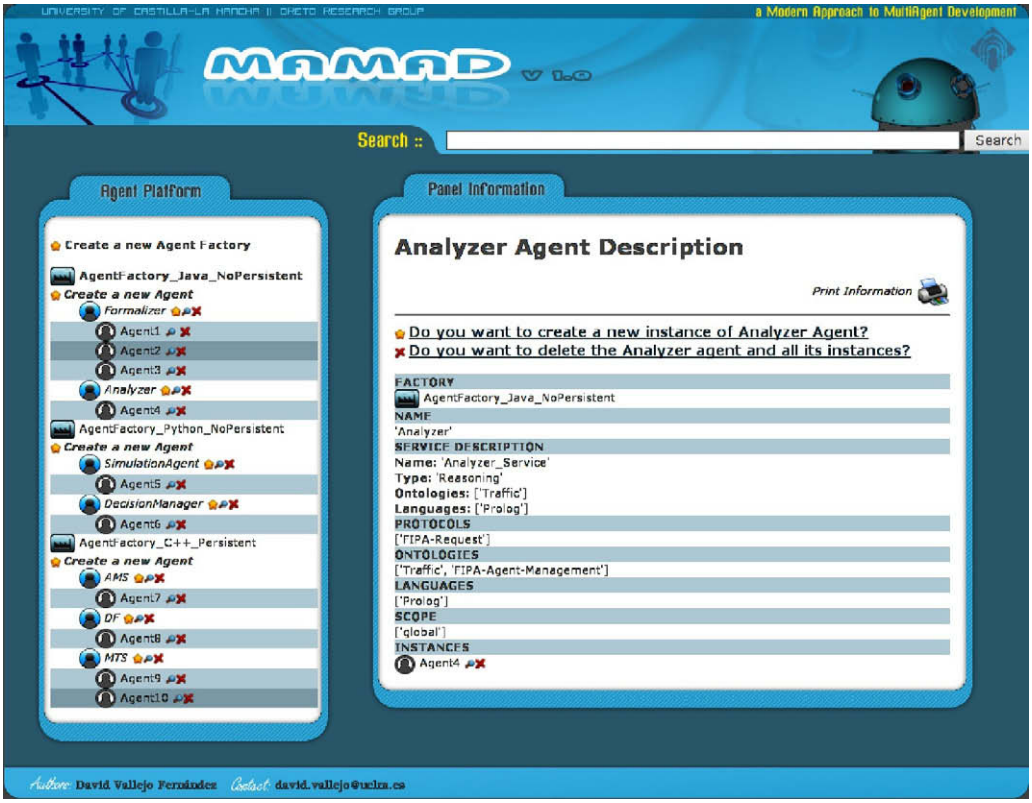

Fig. 7. Administrative web interface of the agent platform.

- Technical questions such as the number of threads per server or the queue size for active agents of each agent factory.

The application deployment and configuration in the agent platform proposed is easy thanks to the use of templates. In fact, we have created service and server templates for all the elements of the agent platform to facilitate this task to developers. Fig. 6 shows some relevant pieces of information of the descriptor created to deploy the multiagent system of Section 4.2.1. Basically, we have servers that host services and both elements can be customised by means of parameters. So, developers only have to instantiate servers in the physical nodes depending on their multiagent applications.

This platform also provides a web application to administrate and deploy MAS, as described in the next section.

3.9. Administrative web interface

Fig. 7 shows a screenshot of the web interface developed to manage and monitor the multiagent platform. In this way, developers have an easy-to-use direct tool to administrate their applications through a web browser, which may help them to interact with the multiagent platform. This interface has been designed and developed for obtaining information about the multiagent platform, such as the existing services, the type of agents, the instances of each one of them, the agent factories, and so on.

The left side of Fig. 7 refers to both the agent factories and the agent instances currently deployed on the user application. This part gives support to the basic agent management, such as the creation, modification or deletion, and basic agent factory management. Depending on the item explicitly selected by the user, the panel of the right side will show detailed information accordingly. For example, if the user clicks on one agent, then the interface will show relevant information of the agent, such as the provided services, the known ontologies, or the supported interaction protocols.

4. Experimental validation

To validate the platform for the development of MAS, we have deployed two sample applications: one to detect anomalous drivers' behaviour in an urban environment (Vallejo et al., 2009) and another to distribute photorealistic rendering (Gonzalez-Morcillo et al., 2007). On the other hand, we have performed a number of tests to measure the efficiency and the scalability of the agent platform. However, the best way to validate it is to invite the reader to download and test it. All the software is freely released[1] under GPLv3.[2]

To both deploy the sample multiagent applications and run the performance tests, the desktop computers with the technical specifications shown in Table 3 have been used. It is important to take into account the number of cores of the computers used to deploy the applications. The evolution of workstations is currently driven by multiprocessing so that developers have to address this issue to obtain high performance systems. Although our system is intrinsically multi-threaded, the final performance is constrained by the number of physical computer cores. That is, the higher the number of cores, the better the performance. In the set of tests carried out in this section we have used typical desktop computers.

We have claimed that one of the main goals of this work is to facilitate the deployment of MAS in many domains as possible. To do that, it is essential to maximise the efficiency and the scalability of the agent platform and to allow developers to use their

[1] http://code.google.com/p/basic-fipa-multiagentsystem.
[2] http://gplv3.fsf.org.

Table 3
Technical specifications of the desktop computers used to test the agent platform.

Operating system	Debian GNU/Linux
Processor	Intel(R) Core(TM)2 CPU
Max CPU Clock	From 2.13 GHz to 2.2 GHz
Cache	4 MB
RAM	From 1 GB to 2 GB

favourite tools (operating systems, programming languages, and hardware platforms). To validate this assertion, we have run two sets of experiments. The first consists in evaluating the platform performance when processing a high number of messages (spamming test). The second measures the scalability by considering the resources consumed by the agent factories (deployment test).

4.1. Examples of practical deployment

In this section we briefly describe two of the MAS that we have deployed thanks to the agent platform presented in this work. The objective is to illustrate how developers can focus their work on designing and programming the agents without worrying about management and communication issues.

The first example refers to the multiagent system shown in Figs. 8 and 9, which aims at detecting and sanctioning anomalous drivers' behaviour in a common urban traffic scenario (Vallejo et al., 2009). This environment refers to a pedestrian crossing which is managed by traffic lights. To reach this global goal, the multiagent system is composed of different agents specialised in several tasks. The *Simulation Agent* simulates the different situations that may take place in the environment, such as a pedestrian who is crossing the road while the vehicles are waiting, a car which is jumping the traffic lights, and so on. This simple reactive agent is implemented in Python. The *Formaliser* is responsible for formalising the information generated by the Simulation Agent. To do that, this agent uses Prolog to formally represent the environment information in a certain time. This agent is implemented in Java. Next, the *Environment Analyzer* is able to distinguish between normal and anomalous drivers' behaviour by using the Prolog inference engine and the previously defined ontology to represent knowledge. This agent is also implemented in Java. Finally, the *Decision Manager* is a reactive agent implemented in Python that registers sanctions if the Environment Analyzer detects an incorrect behaviour.

To deploy this multiagent system, we instantiate two nodes: one to host the FIPA basic services (AMS, DF, and MTS) and one to host the agent factories to create agents. These agents are actually specialisations of the basic agent developed in the platform, so they inherit all the management functions. For instance, the agents are automatically registered with the AMS and the DF and are able to communicate with one another and with the MTS. The communication between agents is made through the FIPA Request Interaction Protocol (Foundation for Intelligent Physical Agents, 2002c), which is supported by the agent platform. So, we only have to specify the content of the messages and what to do in case of error. In order for the agents to discover the services provided by other agents, they use the DF.

The second multiagent application (Gonzalez-Morcillo et al., 2007) refers to a distributed rendering environment. Essentially, this applications is composed of agents specialised in analysing the complexity of the 3D model, managing the whole rendering process and composing the resulting 2D image, and running the rendering of the tasks in which the work was divided. To deploy this multiagent system, we instantiate one node to host the FIPA basic services, one node to host the analyser agent and the manager agent, and additional nodes for each computer used for rendering. Each one of these nodes hosts one, two, or even more rendering agents depending on the number of physical cores of

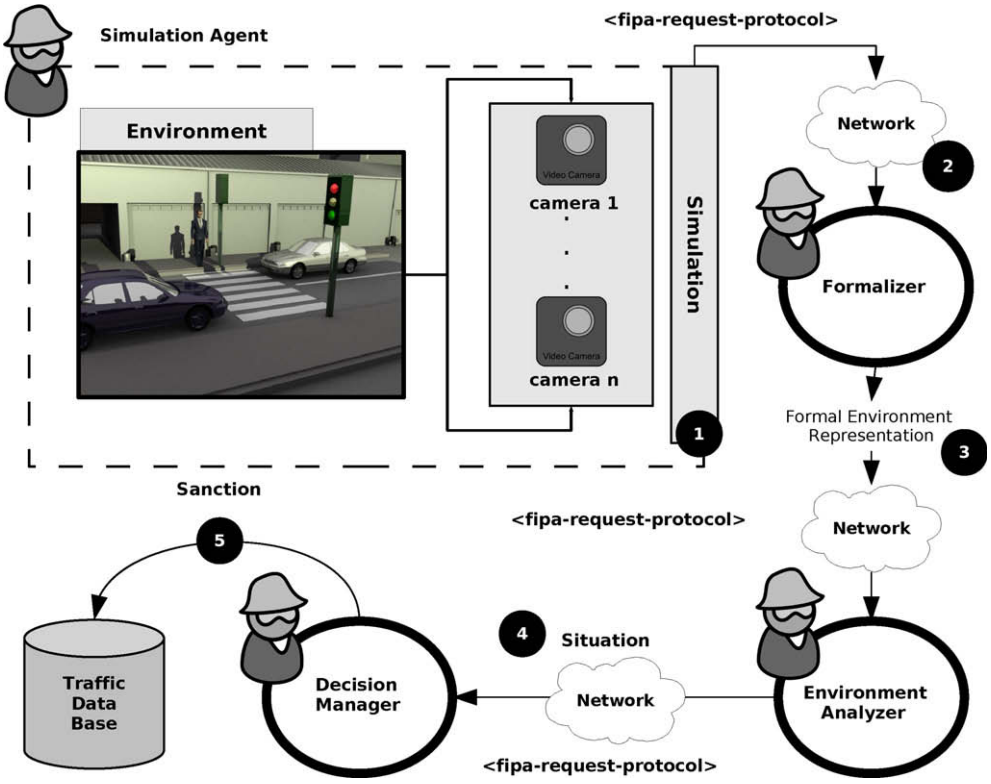

Fig. 8. Example of multiagent application deployed with the platform.

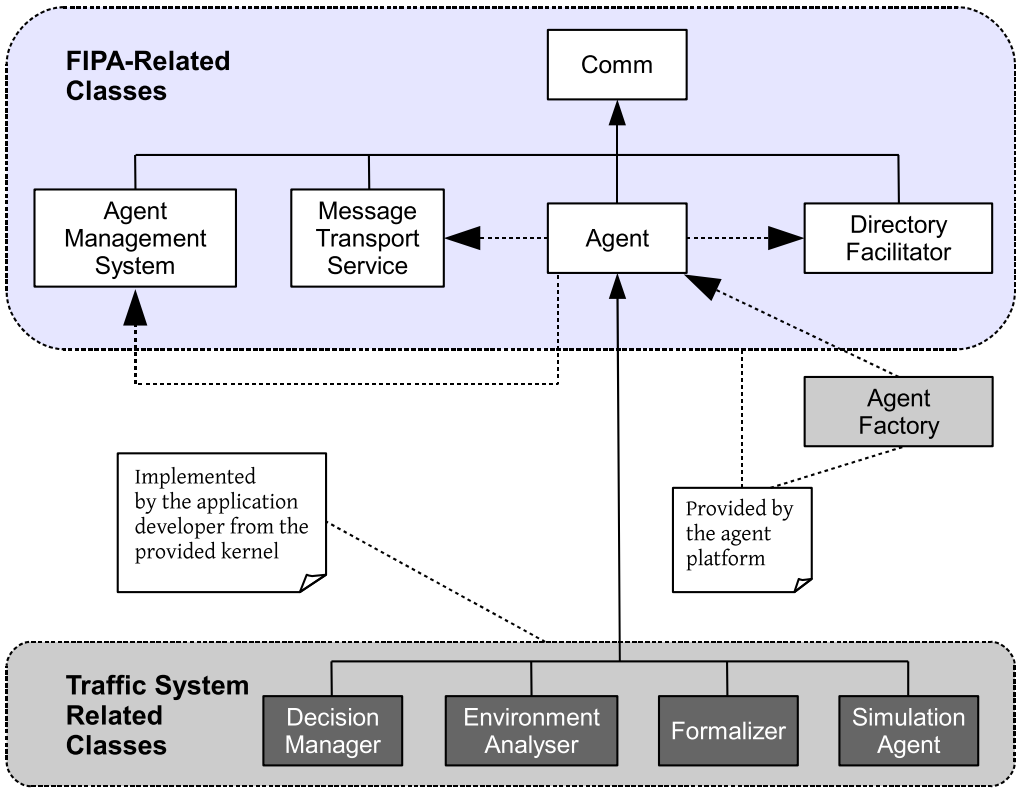

Fig. 9. Class diagram of one of the multi-agent systems developed through the proposed agent platform.

each computer. The communication between agents is carried out without the use of the MTS, and the DF is employed to discover agents depending on their abilities.

4.2. Experimental results

4.2.1. Spamming test

The first series of tests is motivated by the results obtained when evaluating JADE through spamming tests (Chmiel, 2005). Such motivation comes from deploying MAS with JADE in environments that need a high number of agents which communicate with one another.

The architecture used to deploy this scenario is composed of a set computers in which pairs of spammer-user are instantiated. Basically, the spammers send messages to the users, which are responsible for processing them. The number of agent pairs created, the number of messages sent by the spammers, or the message size changes in each test. Furthermore, we measure the time spent in spamming and the time spent by the user agents to process all the messages. The main conclusion obtained in (Chmiel, 2005) is that JADE scales well with the number of agents although the efficiency is constrained due to the use of Java.

To run this series of tests, we have deployed the previously described scenario by using the platform presented in this work, as graphically shown in Fig. 10. This test system has been developed to evaluate any possible test by adjusting different parameters: number of computers, number of nodes, number of spammer agents, number of user agents, number of messages sent by each spammer, and message size. Besides, to make explicit the independence of the agent platform regarding the programming languages used by developers, the spammer agents are implemented in Python and the user agents are implemented in C++.

In order to enrich the JADE evaluation presented in Chmiel (2005), our tests also include the Message Transport Service. In this way, we have measured the efficiency of the message transport from two points of view: (i) direct communication between agents and (ii) indirect communication between agents via the MTS. Obviously, the inclusion of a manager-in-the-middle involves, a priori, a performance decrease. When using the MTS, we have also varied the number of replicas used and the load-balancing policy employed.

The first set of spamming tests in a scenario with a high number of agents is summarised in Table 4.[3] The configuration of this set of tests is as follows:

- Deployment of 3 spammer-user agent pairs in each node. For instance, if the number of nodes is set to 3, then the total number of spammer agents is 9 and the total number of user agents is 9.
- Message size is set to 300 bytes.
- Direct communication between agents, that is, without making use of the MTS.

From these results we underline the following facts (see also Fig. 11a and d). (i) Spamming and processing times increase proportionally to the number of messages and, therefore, to the system load. (ii) The spamming time becomes relevant when the number of messages is from 100,000 on up. However, it is important to take into account that the spammer includes the code needed to attend callback objects as the message sending is carried out asynchronously. (iii) The processing time is longer than one minute when the number of spammers is 9 and the number of messages is 648,000.

The second group of tests sets the message size to 600 bytes and the results are shown in Table 5. The rest of parameters keep their values. The results obtained in this second set of tests confirm the

[3] s stands for seconds.

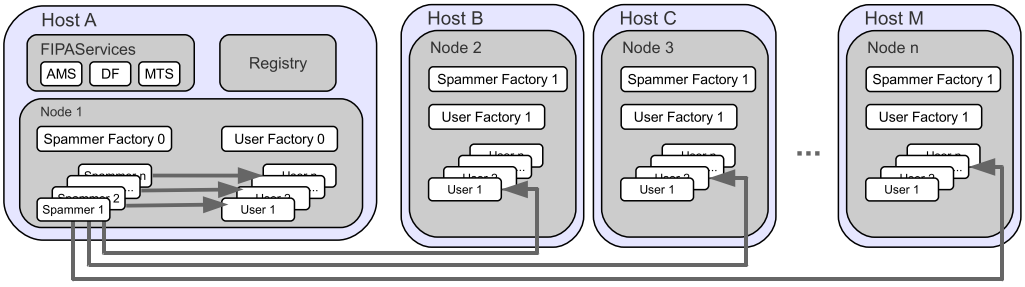

Fig. 10. Scenario to run the spamming tests.

Table 4
First spamming test: 3 agent pairs per node, 300 bytes messages.

Test	Nodes	Spammer messages	Messages	Traffic (MB)	Spamming time (s)	Processing time (s)
1	1	1000	9000	2.57	0.49	0.63
2	2	1000	36,000	10.30	3.73	6.85
3	3	1000	81,000	23.17	5.61	11.25
4	1	2000	18,000	5.14	1.21	1.22
5	2	2000	72,000	20.60	6.38	12.67
6	3	2000	162,000	46.34	10.34	21.51
7	1	4000	36,000	10.30	2.06	2.45
8	2	4000	144,000	41.20	11.97	24.61
9	3	4000	324,000	92.68	20.24	43.08
10	1	8000	72,000	20.60	3.33	5.04
11	2	8000	288,000	82.40	23.16	48.39
12	3	8000	648,000	185.36	38.40	83.58

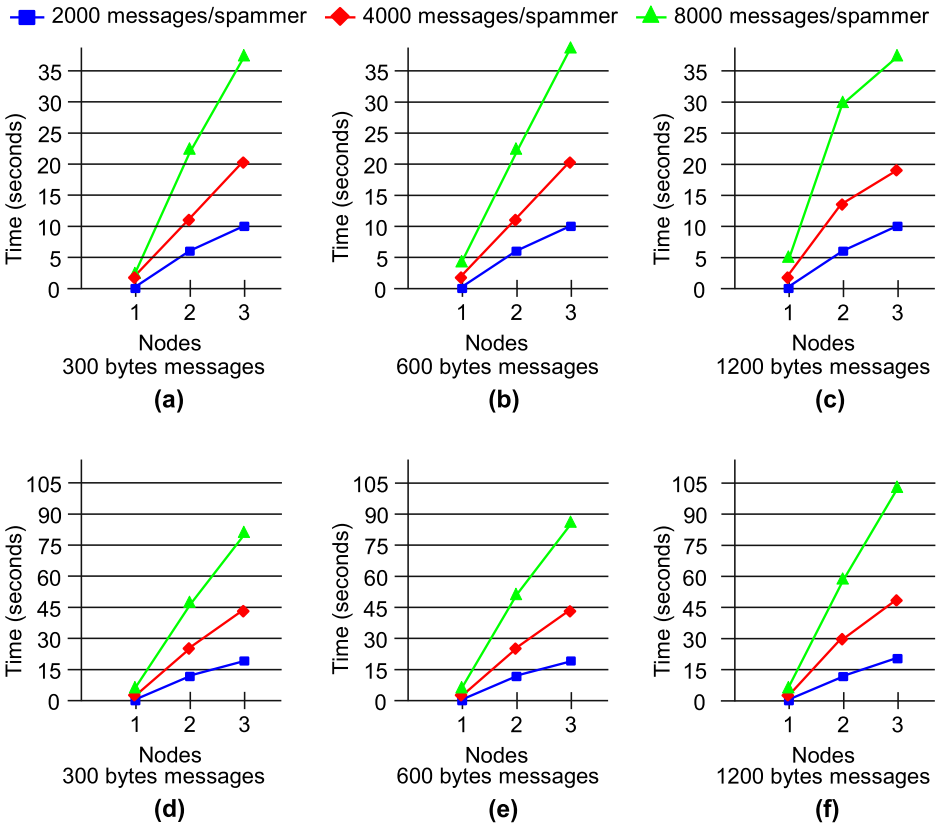

Fig. 11. (a–c). Spamming time. (d–f). Processing time.

conclusions obtained in the first one: scalability of the agent platform regarding the system load and processing time. On the other hand, to double the message size has no significant impact on the resulting times (see Fig. 11b and e).

Table 5
Second spamming test: 3 agent pairs per node, 600 bytes messages.

Test	Nodes	Spammer messages	Messages	Traffic (MB)	Spamming time (s)	Processing time (s)
1	1	1000	9000	5.14	0.60	0.62
2	2	1000	36,000	20.60	3.58	6.88
3	3	1000	81,000	46.34	5.40	11.55
4	1	2000	18,000	10.28	0.83	1.39
5	2	2000	72,000	41.20	6.44	13.01
6	3	2000	162,000	92.68	10.76	22.84
7	1	4000	36,000	20.60	2.33	2.05
8	2	4000	144,000	82.40	11.99	25.07
9	3	4000	324,000	185.36	20.93	44.91
10	1	8000	72,000	41.20	4.25	5.20
11	2	8000	288,000	164.80	23.86	50.17
12	3	8000	648,000	370.72	40.02	88.62

Table 6
Third spam ming test: 3 agent pairs per node, 1200 bytes messages.

Test	Nodes	Spammer messages	Messages	Traffic (MB)	Spamming time (s)	Processing time (s)
1	1	1000	9000	10.28	0.67	0.68
2	2	1000	36,000	41.20	4.24	7.96
3	3	1000	81,000	92.68	5.37	14.34
4	1	2000	18,000	20.56	1.42	1.43
5	2	2000	72,000	82.40	7.71	15.35
6	3	2000	162,000	185.36	9.41	26.78
7	1	4000	36,000	41.20	2.55	2.77
8	2	4000	144,000	164.80	14.71	30.16
9	3	4000	324,000	370.72	18.49	50.01
10	1	8000	72,000	82.40	5.22	5.31
11	2	8000	288,000	329.60	30.18	60.84
12	3	8000	648,000	741.44	38.98	104.36

Finally, the third set of tests sets the message size to 1200 bytes. The results are shown in Table 6. Processing times lightly increase but not in a proportional manner regarding the message size increase (like in the previous set of tests). This fact proves that the system performance does not directly depend on the message size but the number of messages. For instances, the test number 12 of first and third experiment loads the system with 185.36 MB and 741.44 MB, respectively. However, processing times are 83 s and 104 s. That is, data sent is increased by a factor of 4 and processing time by a factor of 1.25. On the other hand, according to spamming times, the message size impact is minimum in all cases (see Fig. 11c and f).

In order to illustrate the scalability of the proposed platform, we have also replicated all these sets of tests with the JADE platform (Bellifemine et al., 2007). This allows us to compare both results and justify the importance of adopting new approaches that contribute to the evolution of agent middlewares. Within this context, Table 7 shows the spamming and processing times spent by JADE under the following assumptions: three agent pairs (spammer-processing agent) per node, two nodes, and a variable number of messages sent and message size.[4] It is important to remark that the sending of messages has also been carried out in an asynchronous way when using JADE.

If we compare, for instance, the times spent when each spammer sends 4000 messages (see test 8 of Tables 4–6 and tests 7, 8, and 9 of Table 7), they are quite different. JADE scales well when running the spamming tests but they take a longer time than the obtained when using the proposed platform. Nevertheless, the most relevant difference in this set of tests can be appreciated when increasing the message size up to 1200 bytes. JADE doubles the processing time with regard to the time spent when sending messages of 600 bytes (see tests 8 and 9 of Table 7) while our platform seems not to depend on the message size to such an extent. Furthermore, JADE processing times are also longer but this might be due to how each platform implements the message reception and the programming language used. The rest of tests corroborate these conclusions.

The second series of tests is on the same line that the first one, in which the agents directly communicate with one another. Nevertheless, we are now interested in using the MTS like message manager. The main goals of these new sets of tests are as follows: (i) to compare previously obtained results with these new ones, (ii) to measure the performance impact of using the MTS with different configurations, and iii) to instruct the reader how to adjust the platform configuration depending on the multiagent system requirements.

Table 8 shows the results obtained under the next assumptions:

- The number of nodes is 3 and each one of them runs in a different computer.
- The number of deployed spammer-user agent pairs is 3 in each node so the total number of pairs is 9 (9 spammer agents and 9 user agents).
- The message size is set to 300 bytes.
- The load-balancing policy between MTS replicas is adaptive, that is, each request is processed by the least-loaded node (CPU cycles).

The results of Table 8 offer interesting conclusions. First, the spamming time is significantly reduced with regard to direct communication between agents. This is because spammers now send one message with multiple receivers to the MTS instead of sending the same message to different user agents. However, the process-

[4] Configuration chosen as a representative case study.

Table 7
JADE spamming test: 3 agent pairs per node, 2 nodes.

Test	Nodes	Spammer messages	Messages	Traffic (MB)	Spamming time (s)	Processing time (s)
1	1000	300	36,000	10.30	1.89	24.77
2	1000	600	36,000	20.60	2.94	31.13
3	1000	1200	36,000	41.20	3.39	38.45
4	2000	300	72,000	20.60	5.67	27.28
5	2000	600	72,000	41.20	7.80	32.50
6	2000	1200	72,000	82.40	12.61	59.73
7	4000	300	144,000	41.20	20.34	29.90
8	4000	600	144,000	82.40	23.34	30.05
9	4000	1200	144,000	164.80	25.72	72.39
10	8000	300	288,000	82.40	44.47	39.84
11	8000	600	288,000	164.80	46.69	91.26
12	8000	1200	288,000	329.60	51.41	156.45

Table 8
Fourth spamming test: 3 nodes, 3 agent pairs per node, 300 bytes messages, communication via the MTS, and adaptive load-balancing policy.

Test	Spammer messages	Messages	Traffic (MB)	Replicas	Spamming time (s)	Processing time (s)
1	1000	81,000	23.17	1	0.57	69.07
2	1000	81,000	23.17	2	0.95	61.65
3	1000	81,000	23.17	3	0.94	47.15
4	2000	162,000	46.34	1	1.17	135.95
5	2000	162,000	46.34	2	1.63	103.15
6	2000	162,000	46.34	3	1.66	95.87
7	4000	324,000	92.68	1	1.99	273.91
8	4000	324,000	92.68	2	2.41	237.02
9	4000	324,000	92.68	3	2.61	197.12
10	8000	648,000	185.36	1	2.85	538.76
11	8000	648,000	185.36	2	3.11	525.87
12	8000	648,000	185.36	3	4.32	246.13

ing time is notably increased, above all in the case of using only one MTS replica to forward messages. In fact, the MTS becomes the bottleneck when processing a high number of messages. The second important issue is that the load-balancing policy employed does not scale well in this experiment. This can be appreciated because the processing time is not considerably reduced when using more than one replica.

Finally, the last test of the spamming experiments is shown in Table 9. In this test we have used a different load-balancing policy based on the round-robin algorithm. In other words, the requests are processed by the different MTS replicas by following a round order.

In this last experiment, the processing times are improved when incrementing the number of replicas (see test 9 of Tables 8 and 9, respectively). The main conclusion after having tested the platform with these configurations is that the developer has to take into account the application domain to choose the correct load-balancing policy.

4.2.2. Deployment test

One of the relevant topics when deploying MAS, specially in domains with computational constraints, is the efficient management of resources. Within this context, one important question is how to manage the physical resources of the different agents that compose the application. By physical resources we mainly mean memory consumption and time spent in changing the state, for instance from hibernating to active.

To address this issue, the most direct solution is to increase the resources. That is, if the number of agents increases, then we can use more powerful hosts, increase their performance level, or use a higher number of them. However, from a practical point of view, it is more interesting to use a mechanism that allows agents to hibernate when they are not needed in order to efficiently manage the host resources. For example, one thousand agents are not practical to reside all in memory at a time, generally. In fact, it makes more sense if a subset of them is in memory and the others are hibernating in persistent storage. To face this challenge, this

Table 9
Fifth spamming test: 3 nodes, 3 agent pairs per node, 300 bytes messages, communication via the MTS, and round-robin load-balancing policy.

Test	Spammer messages	Messages	Traffic (MB)	Replicas	Spamming time (s)	Processing time (s)
1	1000	81,000	23.17	1	0.44	64.95
2	1000	81,000	23.17	2	0.86	50.87
3	1000	81,000	23.17	3	0.85	52.08
4	2000	162,000	46.34	1	1.26	129.51
5	2000	162,000	46.34	2	1.54	105.38
6	2000	162,000	46.34	3	1.29	58.38
7	4000	324,000	92.68	1	2.09	259.25
8	4000	324,000	92.68	2	2.34	212.29
9	4000	324,000	92.68	3	2.17	126.25
10	8000	648,000	185.36	1	3.07	549.17
11	8000	648,000	185.36	2	4.48	478.38
12	8000	648,000	185.36	3	3.89	235.41

Table 10
Agent factory implemented in C++. The queue size of active agents is set to 10.

Agent instances	Persistence	RAM	Creation time (s)	Random requests	Time (s)
100	No	976 KB	0.049	25	0.008
100	Yes	1.7 MB	0.108	25	0.009
1000	No	4.2 MB	0.549	250	0.059
1000	Yes	1.8 MB	2.271	250	0.083
10,000	No	36.6 MB	4.007	2500	0.546
10,000	Yes	1.9 MB	19.494	2500	0.658

Table 11
Agent factory implemented in Java. The queue size of active agents is set to 10.

Agent instances	Persistence	RAM	Creation time (s)	Random requests	Time (s)
100	No	20.8 MB	0.178	25	0.010
100	Yes	22.4 MB	0.304	25	0.022
1000	No	27.8 MB	1.177	250	0.112
1000	Yes	36.3 MB	2.219	250	0.139
10,000	No	90.7 MB	6.581	2500	0.619
10,000	Yes	44.9 MB	25.320	2500	1.009

Table 12
Agent factory implemented in python.

Agent instances	Persistence	RAM	Creation time (s)	Random requests	Time (s)
100	No	3.9 MB	0.072	25	0.007
1000	No	7.3 MB	0.805	250	0.082
10,000	No	40.6 MB	7.764	2500	0.715

platform offers a mechanism to activate agents when needed in order to optimise the use of resources.

The goal of this set of tests is to evaluate this mechanism implemented in the agent factory by measuring the penalty of persistence and agent activation. The tests run involve the instantiation of agents with a state of 1500 bytes and consist in making note of the memory used by servers depending on the number of agents created in a factory. In this way, we can compare the resources consumed by the agent factories with persistence support and by the agent factories with no persistent support. In

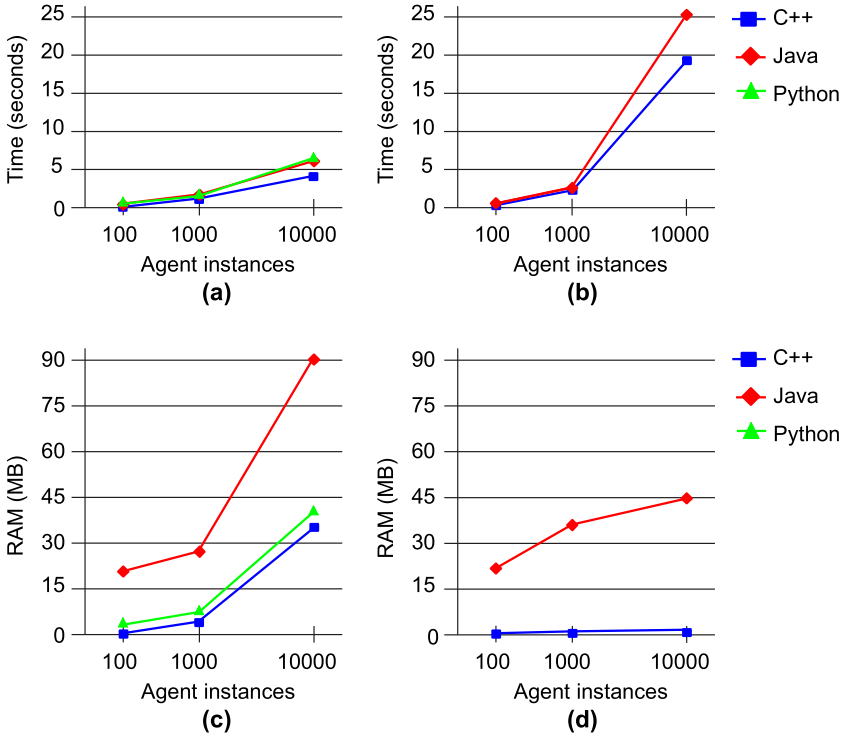

Fig. 12. Top. Time spent when creating agents ((a) with no persistence, (b) with persistence). Below. RAM consumed when creating agents ((c) with no persistence, (d) with persistence).

Table 13
Agent factory implemented in C++. The queue size of active agents is set to 100.

Agent instances	Persistence	RAM	Creation time (s)	Random requests	Time (s)
100	Yes	2.0 MB	0.103	25	0.007
1000	Yes	2.1 MB	1.693	250	0.077
10,000	Yes	2.2 MB	17.536	2500	0.670

Table 14
Agent factory implemented in Java. The queue size of active agents is set to 100.

Agent instances	Persistence	RAM	Creation time (s)	Random requests	Time (s)
100	Yes	26.5 MB	0.304	25	0.020
1000	Yes	28.1 MB	2.509	250	0.148
10,000	Yes	42.5 MB	22.070	2500	0.718

case of using persistence, it is also interesting to obtain the time spent in activating an agent who was hibernating.

Finally, another important parameter of the agent factories with persistence support is the size of the queue of agents in memory. The application developers have to take into account this value in order to obtain high performance. This parameter becomes important when the number of agents is increased. For this reason, and to provide a rigorous evaluation, the experiments cover modifications in this value.

The tests to measure the agent factory performance are run under the following assumptions:

- Agents' state is 1500 bytes.
- Remote invocations over agents is the fourth part of the number of instantiated agents. For instance, if 1000 agents are created, then 250 requests over agents will be randomly done. The idea is to measure the time spent in activating agents which are hibernating.
- The platform to execute the tests was a laptop with a Intel(R) Core(TM)2 CPU 1.8 GHz processor, 2 GB RAM, and a 7200 rpm storage drive. The operating system was Debian GNU/Linux AMD64 with the compiler g++-4.3.2 (with no optimisation), JVM 1.6, and Python 2.5.
- The number of threads of the servers was set to 2 and the communication model was synchronous.

Tables 10 and 11 show the results obtained with the agent factories implemented in C++ and Java, respectively, when the size of the queue of active agents is set to 10. Table 12 shows the results when the agent factory is implemented in Python with no persistence support. The results are graphically summarised in Fig. 12.

Tables 13 and 14 show the results obtained with agent factories implemented in C++ and Java, respectively, when the size of the queue of active agents is set to 100.

Next, we discuss the conclusions obtained after studying the results of this series of experiments with the agent factories. The agent factories that support persistence are slower when instatiating agents but consume less memory when the number of agents is increased. The relevant issue is that the performance impact when reactivating the agents is minimum as request times show. Therefore, and if we take into account the results, the agent factories with persistence support have a great scalability but they need an initial investment when creating the agent database. On the other hand, if the queue size of active agents is increased, the times are reduced but RAM use is lightly higher. Thus, the developers must evaluate again different configurations before deploying the final multiagent system. Finally, the Java version uses much more memory than the C++ version, but it scales well in processing time and memory use.

5. Conclusions

We have presented a modern agent platform to develop MAS according to FIPA specifications. The main contribution of this paper is to provide a new approach extensible to a wide range of platforms and domains when developing MAS. To do that, we have designed and implemented a software architecture based on design patterns and templates on top of a modern middleware which covers a high number of operating systems, programming languages, hardware devices, and communication networks. The main characteristics of this agent platform are as follows:

- Development based on the set of FIPA standards.
- Automatic deployment of FIPA services, which are highly configurable depending on the application constraints.
- Use of a modern middleware interoperable with a high number of programming languages, operating systems, and hardware platforms.
- Ready-to-use agents developed in different programming languages.
- Integration with an administrative web system to remotely manage multiagent applications.
- Scalability, robustness, authentication, encryption, and message integrity.

To evaluate our proposal, we have carried out an extensive set of experiments to mainly test the efficiency and the scalability of the agent platform. Results show that the system offers good response-times when the load of the multiagent application becomes relevant and suggest that this kind of tools must be as configurable as possible to adequate the system deployment to the application requirements. Besides, two multiagent application examples have been described in order to show the agent platform features and how the development of new tools can help to deploy and manage MAS.

We are continuing the development of the agent platform. Our main goal is to interoperate with JADE for agents to interact one another without worrying about the underlying technology. To do that, we are developing a gateway agent between our platform and JADE due to the use of different transport protocols. Moreover, we are currently evaluating how to include mobility support for the agents to move out between hosts. For example, we are very interested in including a mechanism that allows a concrete agent

to migrate to a certain host if more resources are needed to attend a request. To do that, we are designing a mobility protocol which establishes the rules to be followed in order for agents to clone or migrate.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments that have significantly improved the quality of this paper. This work has been founded by the Regional Government of Castilla-La Mancha under the Research Project PII1C09-0137-6488 and by the Ministry of Science and Innovation under the Research Project TIN2009-14538-C02-02 (FEDER). Special thanks to Ignacio Arriaga for his support in the implementation of the JADE spamming tests.

References

Bellifemine, F.L., Caire, G., Greenwood, D., 2007. Developing Multi-agent Systems with JADE. Springer.
Bellifemine, F.L., Caire, G., Poggi, A., Rimassa, G., 2008. JADE: A software framework for developing multi-agent applications. Lessons Learned, Information and Software Technology, Elsevier 5 (1–2), 10–21.
Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A., 2006. A survey of programming languages and platforms for multi-agent systems. Informatica 30 (1), 33–44.
Chmiel, K., 2005. Efficiency of JADE agent platform. Scientific Programming 13 (2), 159–172.
Downing, T.B., 1998. Java RMI: Remote Method Invocation. Wiley Publishing.
Foundation for Intelligent Physical Agents, 2002a. FIPA Abstract Architecture Specification. <http://www.fipa.org/specs/fipa00001>.
Foundation for Intelligent Physical Agents, 2002b. FIPA Agent Message Transport Service Specification. <http://www.fipa.org/specs/fipa00067>.
Foundation for Intelligent Physical Agents, 2002c. FIPA Request Interaction Protocol Specification. <http://www.fipa.org/specs/fipa00026>.
Foundation for Intelligent Physical Agents, 2004. FIPA Agent Management Specification. <http://www.fipa.org/specs/fipa00023>.
Foundation for Intelligent Physical Agents. <http://www.fipa.org>, last checked: 14/01/09.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading.
Gonzalez-Morcillo, C., Weiss, G., Jimenez, L., Vallejo, D., 2007. A Multi-Agent Approach to Distributed Rendering Optimization. Innovative Applications of Artificial Intelligence Conference (IAAI 2007), pp. 1775-1780.
Helsinger, A., Wright, T., 2005. Cougaar: a robust configurable multi agent platform. IEEE Aerospace Conference, 1–10.
Henning, M., 2004. A new approach to object-oriented middleware. IEEE Internet Computing, IEEE Computer Society 8 (1), 66–75.
Henning, M., 2006. The rise and fall of CORBA, ACM Queue. ACM New York 4 (5), 28–34.
Huhns, M.N., Singh, M.P., 2005. Service-oriented computing: key concepts and principles. IEEE Internet Computing, IEEE Computer Society 9 (1), 75–81.
Jain, P., Schmidt, D.C., 1997. Dynamically configuring communication services with the service configurator pattern. C++ Report 9 (6), 29–42.
Jennings, N.R., 2000. On agent-based software engineering. Artificial Intelligence, Elsevier 117 (2), 277–296.
Odell, J., 2000. Agent Technology, OMG, green paper produced by the OMG Agent Working Group.
OMG Agent Platform Special Interest Group. <http://agent.omg.org>, last checked: 14/01/09.
Pokahr, A., Braubach, L., Lamersdorf, W., 2005. Jadex: A BDI reasoning engine. Multiagent Systems Artificial Societies and Simulated Organizations, vol. 15. Springer. pp. 149–174.
Ricci, F., Nguyen, Q.N., 2007. Where are all the intelligent agents. IEEE Intelligent Systems, IEEE Computer Society 22 (3), 2–3.
Ross, R., Collier, R., O'Hare, G.M.P., 2004. AF-APL-bridging principles and practice in agent oriented languages. In: Second International Workshop on Programming Multi-Agent Systems (ProMAS'04). Lecture Notes in Computer Science, vol. 3346. Springer, pp. 66–88.
Simple Object Access Protocol Messaging Framework. <http://www.w3.org/TR/soap12-part1>, last checked: 14/01/09.
Vallejo, D., Albusac, J., Jimenez, L., Gonzalez, C., Moreno, J., 2009. A cognitive surveillance system for detecting incorrect traffic behaviors. Expert Systems with Applications, Elsevier 36 (7) 10503–10511.
Villa, D., Villanueva, F.J., Moya, F., Rincon, F., Barba, J., Lopez, J.C., 2007. Minimalist object oriented service discovery protocol for wireless sensor networks. Lecture Notes in Computer Science, Springer (4459/2007), 472–483.
Weiss, G., 1999. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press.
Winikoff, M., 2006. JACK TM intelligent agents: an industrial strength platform. Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer. pp. 175–193.
Wooldridge, M., 2001. An Introduction to Multiagent Systems. John Wiley & Sons, Inc., New York, NY, USA.
Wooldridge, M., Ciancarini, P., 2001. Agent-Oriented Software Engineering: The State of the Art. Lecture Notes in Computer Science, Springer (1951/2001), 1–28.
Wooldridge, M., Jennings, N.R., 1995. Intelligent agent: theory and practice. Knowledge Engineering Review, Cambridge University Press 10 (2), 115–152.
ZeroC Freeze Evictor Documentation. <http://zeroc.com/doc/Ice-3.3.0/manual/Freeze.37.5.html>, last checked: 14/01/09.
ZeroC ICE Performance Analysis. <http://zeroc.com/performance/index.html>, last checked: 14/01/09.

David Vallejo is a Lecturer and a Ph.D. candidate in the ORETO research group at the University of Castilla-La Mancha. His recent research topics are multi-agent systems, cognitive surveillance architectures, and distributed rendering. He received the B.S. and M.S. degrees in Computer Science from the University of Castilla-La Mancha in 2006 and 2008, respectively.

Javier Albusac is a Lecturer and a Ph.D. candidate in the ORETO research group at the University of Castilla-La Mancha. His recent research topics are multi-agent systems, cognitive surveillance, and scene understanding. He received the B.S. and M.S. degrees in Computer Science from the University of Castilla-La Mancha in 2005 and 2008, respectively.

Jose Angel Mateos is a junior researcher and a Ph.D. candidate in the ORETO research group at the University of Castilla-La Mancha. His recent research topics are multi-agent systems, virtual reality architectures, and distributed rendering. He received the B.S. degree in Computer Science from the University of Castilla-La Mancha in 2005.

Carlos Gonzalez is a Reader at the University of Castilla-La Mancha. His recent research topics are multi-agent systems, virtual reality architectures, and distributed rendering. He received the B.S. and Ph.D. degrees in Computer Science from the University of Castilla-La Mancha in 2002 and 2007, respectively.

Luis Jimenez is an Associate Professor of Computer Science at the University of Castilla-La Mancha. His recent research topics are multi-agent systems, knowledge representation, ontology design, and fuzzy logic. He received the M.S. and Ph.D. degrees in Computer Science from the University of Granada in 1991 and 1997, respectively. He is member of the European Society of Fuzzy Logic and Technology.