DOI: 10.2298/CSIS111029039A

A Scalable Multiagent Platform for Large Systems

Juan M. Alberola, Jose M. Such, Vicent Botti,
Agustín Espinosa and Ana García-Fornes

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València Camí de Vera s/n. 46022, València (Spain)
{jalberola,jsuch,vbotti,aespinos,agarcia}@dsic.upv.es

Abstract. A new generation of open and dynamic systems requires execution frameworks that are capable of being efficient and scalable when large populations of agents are launched. These frameworks must provide efficient support for systems of this kind, by means of an efficient messaging service, agent group management, security issues, etc. To cope with these requirements, in this paper, we present a novel Multiagent Platform that has been developed at the Operating System level. This feature provides high efficiency rates and scalability compared to other high-performance middleware-based Multiagent Platforms.

Keywords: Multiagent Platforms, Multiagent Systems, Evaluation.

1. Introduction

In the last decade, due to the rapid growth of the Internet, the speed of change, and an ever greater amount of easily accessible information, the next generation of Multiagent Systems (MAS)s and information technology, will target open and large systems. In these dynamic and heterogeneous environments, it is essential that features such as security, high performance, scalability, and interoperability are provided by application development frameworks.

Even though current Multiagent Platforms (MAP)s support the development and execution of MASs, very few real applications have been developed to focus on open and dynamic systems. These applications change quickly and require features such as reliability, scalability, and performance, which not many MAPs are designed to offer. According to [25], agent researchers should design and implement large software systems consisting of hundreds of agents and not only systems composed of a few agents. In order to develop these systems, researchers require efficient and scalable MAPs.

Some current MAPs are not suitable for executing complex systems because their designs are not oriented to improving efficiency and scalability issues. Previous studies have demonstrated a degradation in the performance of current MAPs as the system grows [51, 22]; some MAPs even fail [49]. Our main objective for this paper is to propose a MAP that is focused on being scalable and efficient. One of our main design decisions is to use the operating system (OS) services to develop this MAP instead of using middlewares between the OS and the MAP. In [14] we proved that this can noticeably improve the performance and scalability of the system.

Functionality is another important issue when executing large systems. Works by other researchers such as [20] are helpful in determining the main requirements for designing a MAP. By using theoretical proposals and methodologies [27], a MAP that supports agent organizations helps to simplify, structure, coordinate, and easily develop large applications, which are composed of thousands of agents. Standard language communication is another key requirement for allowing the interaction between heterogeneous entities. Support to coordinate communication is another requirement for these systems [42]. Definition of standard speech acts that agents can use, a common ontology to describe and access services, policies associated to agent conversations, and standard communication language are some features that should be provided. Finally, security concerns become important in large systems must be addressed if these systems are open in order to ensure the communications and the identities of each entity. As stated by other authors in [45], these features should be provided by agent execution frameworks.

Towards these goals, in this paper, we present a MAP that is oriented to fulfilling the requirements for this new kind of systems. This MAP is mainly focused on scalability and efficiency for executing large MASs. It provides mechanisms to support agent organizations, security concerns (authentication, authorization, and integrity), a standard language of communication for information representation, conversation-oriented interactions, and so on.

The rest of the article is organized as follows. Section 2 presents the motivation and the previous work that allowed us to design and develop an efficient and scalable MAP. Section 3 gives an in-depth description of the MAP architecture. Section 4 details the services offered by the MAP. Section 5 describes how agents in this MAP are represented. Section 6 describes a tourism service application that is built on this MAP. Section 7 presents a performance evaluation of the MAP. And finally, in Section 8, we present some concluding remarks.

2. Motivation and previous work

In the last few years, many researchers have focused on testing the performance of existing MAPs. One of the main properties tested in these works is the performance of the MAPs for sending messages. Vrba [51] presents an evaluation of the messaging service performance of four MAPs. From the tests presented in that paper, the author concludes that Jade [19] provides the most efficient messaging service compared to FIPA-OS [1], Jack [3], and ZEUS [12]. However, the design features that produce this performance are not given and the implementations of the messaging service for each MAP are not detailed. Therefore, these conclusions can only be valid to choose the MAP that performs better than the other three MAPs tested. Burbeck et al. [22] tested the messaging service performance of three MAPs. They claim that Jade performs better than Tryllian [11] and SAP [9] because it is built on Java RMI[1], but they give no proofs confirm this claim. As these works state, Jade is more scalable than other MAPs and can be considered to be a stable MAP for large systems [40]. However, these conclusions do not provide any clue to MAP developers about how to improve MAP designs since these experiments only scale up to 100 pairs of agents and a few hosts. A more thorough study is required to be able to assess MAP performance and to determine to what extent design decisions influence MAP performance.

Some other works have tested the performance of other services but only for a single MAP. Most of these works test Jade, which seems to be the most widely used MAP. In [25], the authors tested Jade messaging, agent creation, and migration services. The tests that they performed related to the messaging service only scale up to eight agent pairs. In [17], an evaluation of a MAS for adapting application's behaviour was carried out on Jade MAP. The work presented in [26] tested the scalability and performance of the Jade messaging service. Similar to the works cited above, their conclusions do not provide any design decision. Even though these conclusions can allow MAS developers to check whether or not Jade fulfills their requirements when designing a MAS, they do not suggest any design decision for MAP developers.

There are also other works that focus on testing the performance of a specific MAS that is running on top of a MAP. In [23] the performance of MAPs is measured when a MAS composed of several web agents is launched. This MAS provides documents requested by a user agent. The authors measured the number of documents requested per unit of time. Therefore, their conclusions are only valid for this MAS. Lee et al. [37] present a MAS in which agents coordinate with each other to carry out tasks. They evaluate how the topological relations between agents affect the number of CPU cycles needed to accomplish these tasks. In [28], the authors compare the response time and the CPU cycles of SACI [13] and Jade.

Finally, other studies focus on detailing the functional properties of MAP. In [20], four MAPs are compared according to several criteria: implementation languages, tools provided, agent deliberation capabilities, etc. Shakshuki [46] presents a methodology to evaluate MAPs based on several criteria such as availability, environment, development, etc. Similar work is carried out by Nguyen [33], and Omicini [43] gives a brief evolution of MAPs. In other works such as [34, 44], different MAPs that are intended to be scalable are proposed; however, no empirical evaluation is carried out. These works provide ratings of properties provided by MAPs in order to help users choose the MAP according to their needs. Our work goes a step further since it is not only intended to be useful for MAP users but also for MAP developers.

A general conclusion of works that focus on MAP evaluation is that MAP performance decreases as the system grows. Furthermore, as we showed in a previous work [49], when large-scale MAS are taken into account, the performance of many MAPs is considerably degraded when the size of the system

[1] http://java.sun.com/docs/books/tutorial/rmi/index.html

executed increases, causing some MAPs to even fail. Therefore, current MAPs are not suitable for executing large population systems because their designs are not aimed at improving efficiency and scalability issues.

In order to develop a design in accordance with our goals, we detail other previous works that we carried out that focus on finding design decisions that influence MAP performance. In [41], we presented experiments to link performance with internal MAP designs, that is, to identify the key design decisions that lead to better performance. We extracted some conclusions from these experiments, such as the fact that centralizing services in a single host of the MAP degrades the performance causing this host to become a bottleneck in the case of very popular services. It is more suitable to design a distributed approach with efficient information replication mechanisms. In [16], we tested several issues of the MAPs, such as the performance of the directory service proposed by FIPA [2], the memory consumed by the agents and the MAP, the network occupancy rate, the CPU cycles, etc. According to these studies, the most influential point in the MAP performance that could become a bottleneck is the messaging service. This service is crucial in the performance of the MAP since agents need to exchange messages with other agents and access MAP services. Furthermore, some MAPs (such as Jade) base other MAP services (such as the Agent Directory or Service Directory proposed by FIPA) on the messaging service. In [14], we specifically analyzed technologies for implementing the Message Transport System (MTS), which is the component of the MAP that manages the message exchanges among the agents running on the MAP. This work showed that in order to design a messaging service that can handle large agent populations, the design that performs better should be based on direct communication between each pair of agents so that the messaging service scales better and performs more efficiently, especially in these sorts of scenarios.

In the following sections, we present a MAP focused in being scalable and efficient in more detail. It has been developed using the services offered by the OS to support MAS efficiently. By bringing MAP design closer to the OS level we can define a long-term objective, i.e., to incorporate the agent concept into the OS itself in order to offer a greater abstraction level than current approaches.

3. Magentix Multiagent Platform architecture

Magentix[2] MAP aims to be scalable and efficient, mainly when it is executing large-scale MAS. To achieve a response time closer to the achievable time lower bound, this MAP has been developed using the services provided by the OS. Thus, one of the design decisions is that this MAP is written in C over the Linux OS. Current approaches for developing MAPs are based on interpreted languages like Java or Python. These MAP designs are built over middlewares like the Java Virtual Machine (JVM) [21]. Although these middlewares offer some advantages like portability and easy development, MAPs developed over them

[2] Magentix can be downloaded from http://gti-ia.dsic.upv.es/sma/tools/Magentix/index.php

do not perform as well as one might expect, especially when they are running large systems. In [14] we presented a performance evaluation related to this issue. We proved in that using the Operating System (OS) services to develop a MAP instead of using middlewares between the OS and the MAP noticeably improves the performance and scalability of the MAP. Thus, we can see the MAP functionality as an extension of the functionality offered by the OS.

The Magentix communication service has been developed to offer high performance. This service is quite crucial to the performance of the MAP as we stated in Section 2 and some other services may be implemented using it. Magentix also provides advanced communication mechanisms such as agent groups, a manager to execute interaction protocols, and a security mechanism to provide authentication, integrity, confidentiality, and access control. This design has been developed in order to provide the functionality required by MAS and perform efficiently.

Magentix is a distributed MAP composed of a set of computers executing Linux OS (figure 1). Magentix uses replicated information on each MAP host to achieve better efficiency. Each one of these computers presents a process tree structure. The initial design of this structure is presented in [15]. The advantage of process tree management offered by Linux, and using some services like signals, shared memory, execution threads, sockets, etc. provides a suitable scenario for developing a robust, efficient, and scalable MAP.

The structure of each Magentix host is a three-level process tree. On the higher level we see the *main* process. This process is the first one launched on any host when this host is added to the MAP. Below this level we can see the services level. Magentix provides some services to support agent execution: Agent Management System (*AMS*), Directory Facilitator (*DF*), and Organizational Unit Manager (*OUM*). Services are represented by means of service agents replicated in every MAP host. Agents representing the same service manage replicated information and communicate with each other in order to keep this information updated. Finally, in the third level, user agents are placed. Using this process tree structure, *main* process manages service agents completely, i.e., it can kill any service agent to achieve a controlled shutdown of the MAP, and also detects at once whether any service agent dies. In the same way, *ams* agent has a broad control of the user agents of its own host.

Each user agent is represented by a different Linux child process of the *ams* agent running on the same host. This design decision was taken after efficiency tests as we stated in Section 2. Mapping one-to-one agents and Linux processes provides us with a complete execution control (as we will see in the next section) and a fast message exchanging mechanism. It could be argued that using a single virtual machine for executing agents represented as Java threads could be lighter. Nevertheless, this virtual machine could be overloaded when running three or four thousand agents, by the limitations of the virtual machine. In our proposal, mapping agents as Linux processes restricts us to the limitations of the OS, and allows us to run more than seven thousand agents in a single host. Developing a MAP by using the OS services directly allow us to

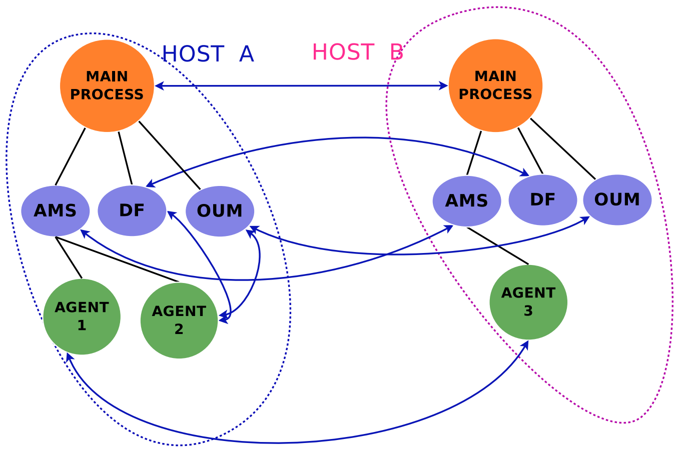

Fig. 1. Platform structure: Agent Management System (*AMS*), Directory Facilitator (*DF*), Organizational Unit Manager (OUM)

improve the efficiency of the system. Current Magentix version offer support to different Linux distributions (such as Ubuntu, Fedora, CentOs or OpenSuse) as well as to Mac OS. Interoperability between heterogeneous agents is reached by means of standard communication language representation and ontologies for service interactions.

3.1. Communication and Message Transport System

Magentix provides a message-based communication mechanism in order to allow interactions between agents and services. This communication mechanism aims to obtain both good efficiency level and MAP scalability. As Magentix MAP is integrated into Linux, we have checked different alternatives available for communicating processes in an OS context [14]. In this study we have analyzed different communication services among processes provided by POSIX [10] compliant OS, in particular, the Linux OS, in order to select which of these services allows robust, efficient and scalable MAPs to be built. As a result of the evaluation, a lower bound of the time needed to communicate process couples (located in the same or different hosts) was obtained. In these studies, we showed to what extent the performance of a Message Transport System (MTS) degrades when its services are based on middlewares between the OS and the MAP (like the JVM) rather than directly by the underlying OS. Thus, the Magentix MTS design was tested to be as close as possible to this time lower bound.

As we pointed out in section 2, the messaging service design that should perform better would be one based on direct communication between each pair of agents. Therefore, the communication mechanism implemented in message exchanging interactions is carried out by means of point to point connections based on TCP sockets, between a pair of processes. This mechanism enables high scalability in agent communication. Each Magentix agent has a server socket for receiving connections from other agents by means of client sockets. To carry out a new connection an agent creates a client socket that communicates with the remote agent server socket. Thus, Magentix agents are client/server at the same time.

At a lower level, Java-RMI technology (used for development communication in most of MAPs based on Java) uses TCP sockets. After evaluating different alternatives, we finally define the communication mechanism implemented in Magentix as point to point connections based on TCP sockets, between a pair of processes. The use of C language to develop the MAP, allows us to use this technology closer to the OS level, and avoid the overhead resulting from the use of Java-RMI, because the agent abstraction provided by a MAP is independent of the underlying communication mechanism implementation.

In our previous studies, we have also checked that opening a P2P connection between a pair of agents the first time they interact and leaving this connection open for future interactions is much more efficient than opening a new TCP connection each time they want to interact. Therefore two agents could have an indefinitely open connection for exchanging messages each time they require it. Nevertheless, the number of simultaneous open connections is limited by the OS. Therefore, each agent and service stores its open connections in a *connection table*. The first time an agent contacts another one, a TCP connection is established and remains open to exchange messages in the future. These connections are automatically closed when the conversation is not active, that is, some time has passed since the last message was sent, according to a LRU (Last Recently Used) policy (this mechanism is described in more depth in [50]). This *connection table* improves communication times since an agent does not need to create a new TCP connection each time it wants to communicate with another agent.

4. Services

In this section we describe the services that are implemented in Magentix oriented to agents, services, and group management: *AMS* service, *DF* service and *OUM* service.

4.1. Agent Management System

Agent Management System (*AMS*) service is defined by FIPA [29] and offers the white pages functionality. This service stores the information regarding the agents that are running on the MAP. *AMS* service is distributed among every MAP host. Therefore, information regarding the agents of the MAP is replicated in each host. This service is represented as *ams* agents running in each host of the MAP.

As we stated in section 3, all of the agents launched in a specific host are represented by means of child processes of the *ams* agent. Just as the *main* process behaves, the *ams* agent has a broad control of the agents in its corresponding host. The management of starting and finalizing agents is automatically carried out by means of sending signals

The *AMS* service stores the information regarding every agent running on the MAP. This service allows us to obtain the physical address (IP address and port), providing the agent name to communicate with. Due to the fact that the *AMS* service is distributed among every MAP host, each *ams* agent running on each host contains the information needed to contact every agent of the MAP. Hence, the operation of searching agent addresses is not a bottleneck as each agent looks this information up in its own host, without needing to make any requests to centralizing components. Every time an agent is started or finalized in a host, this update is replicated on each host of the MAP. Nevertheless, there is another information regarding agents that does not need to be replicated when it is updated. For this reason, the *ams* agents manage two tables of information: the Global Agent Table (*GAT*) and the Local Agent Table (*LAT*).

– **GAT**: Stored in this table is the name of each agent in the MAP and its physical address, that is, its IP address and its associated port.
– **LAT**: In this table additional information is stored such as the agent's owner, the process PID which represents each agent and its life cycle state.

The *GAT* is mapped on shared memory. Every agent has read only access to the information stored in the *GAT* of its own host. Each time an agent needs to obtain the address of another agent in order to communicate, it accesses the *GAT* without making any request to the *ams* agent. Thus, we avoid the bottleneck of requesting centralizing components each time one agent wants to communicate with another. The information contained in the *GAT* needs to be replicated in each host to achieve better performance. Although replication mechanisms imply an overhead in the system, this overhead is reduced as only the updated information is replicated, and these updates occur when agents are started or dead in the MAP, operations that generally occur in low frequency rates. Thus, the overhead resulting from replication is worthwhile in order to distribute the information and make it available in each host of the MAP. Moreover, the spacial overhead (memory) for having the same information replicated in each host is also low, due to the fact that only the physical addresses of the agents are distributed (few bytes of memory).

The information from the *LAT* is not replicated. Some information stored in the *LAT* regarding a specific agent is only needed by the *ams* agent of the same host (for instance, the process PID). Therefore, this information does not need to be replicated. Some other information could be useful for the agents but is not usually requested (such as the life cycle state). In order to reduce the overhead resulting from replication, we divide the information regarding agents into two tables. Each *ams* stores in their *LAT* the information regarding the agents under its management, that is, the agents that are running on the same host. If some information available to agents is needed (such as the life cycle state), the agent has to make a request to the *AMS* service using the *AMS* service ontology. In a transparent way, these requests addressed to the *AMS* service are delivered to the specific *ams* running on the same host as the agent requested.

4.2. Directory Facilitator

The Directory Facilitator (*DF*) service offers the yellow pages functionality defined by FIPA. This service stores the information regarding the services offered by agents. The *DF* service allows agents to register the services they provide, deregister these services, and look up a specifically required service. Much like the *AMS* service, the *DF* service is implemented in a distributed scenario by means of agents running on each MAP host, called *df* agents. Information regarding services is also replicated in every host of the MAP.

Information that needs to be replicated is stored in a unique table called **GST** (Global Service Table). This table is a list of pairs: services offered by agents of the MAP and the agent that offers this service. In contrast to the **GAT**, the **GST** is not implemented as shared memory, therefore only the *df* can acces this information directly.

Agents are able to register, deregister, and look up services offered by other agents. To do these tasks, agents need to communicate with the *DF* service using the *DF* service ontology. Current functionality of the *DF* service is the one proposed by FIPA. Nevertheless, we consider the possibility of improving this service in order to provide new functionalities such as the management of semantic information, service composition, services offered by agent organizations, etc. and also extending the operations proposed by FIPA for registering, deregistering, and searching for services.

4.3. Organizational Unit Manager

The Organizational Units Manager (*OUM*) service provides support oriented to agent-group communication as a pre-support for agent organizations. Several research groups define theoretical proposals and methodologies to design MASs, oriented to organizational aspects of the agent society [27]. In order to develop applications which use these organization oriented methodologies, we require MAPs that support them. Among there are few MAPs which offer any kind of support related to agent organizations. Among these MAPs are Jack [3], MadKit [4], or Zeus.

An agent group in Magentix is called organizational *unit* (from now on, unit) and can be seen as a blackbox from the point of view of external agents. Units can also be composed of nested units. Agents can interact with an agent unit in a transparent way, i. e. from the point of view of an agent outside the unit, there is no difference between interacting with a unit or with an individual agent. Interaction between an agent and a unit is carried out by the MAP through properties specified by the user. Each unit has some properties associated to it. As each agent of the MAP has a unique name, each unit is identified in the MAP by its *name*. In order to interact with any unit, user must specify one or more agents to receive the messages addressed to the unit: these agents are called *contact agents*. User can also specify the way in which these messages have to be delivered to the *contact agents*. This property is called the *routing type* and messages addressed to the unit will be delivered to the contact agents defined according to one of these *routing types*:

- Unicast: The messages addressed to the unit are delivered to a single agent which is responsible for receiving messages. This type is useful when we want a single message entrance to the group. It could be useful if the group has for example, a hierarchical structure, where the supervisor receives every message and distributes them to its subordinates.
- Multicast: Several agents can be appointed to receive messages. When a message is addressed to the unit, this message is delivered to any contact agent in the unit. This could be useful if we want to represent an anarchic scenario, where every message needs to be known by every agent without any kind of filter.
- Round Robin: There can be several agents appointed to receive messages. But each message addressed to the unit is delivered to a different contact agent, defined according to a circular policy. This type of routing messages is useful when several agents offer the same service but we want to distribute the incoming requests to avoid the bottlenecks.
- Random: Several agents can be defined as contact agents. But the message is delivered to a single one, according to a random policy. As with the previous type, this is useful for distributing the incoming requests, but no kind of order for attending these requests is specified.
- Sourcehash: Several agents can be the contact agents. But any given message is delivered to one of the agents responsible for receiving messages, according to the host where the message sender is situated. This is a load-balancing technique.

Units have a defined set of agents which make up the unit, called *members*. These agents can interact and coordinate with each other and each one plays a certain *role*. Finally, each unit has a *manager* associated to it. This agent is responsible for adding, deleting or modifying the members and contact agents. By default it is the agent which creates the unit and is the only one allowed to delete it.

All of this information regarding units in Magentix, is managed by the *OUM* service, which stores it in the **GUT** (Global Unit Table). Similar to the previous services, *OUM* is a distributed service composed by *oum* agents running on each MAP host. The **GUT** table is replicated and synchronized on each host of the MAP every time an update is made. Interaction between agents and *OUM* service is carried out by the sending of messages using the *OUM* service ontology.

4.4. RDF as framework for representing information

To develop large systems, standard language communication is a key requirement for allowing the interaction between heterogeneous entities. FIPA proposes some Agent Communication specifications regarding the language used for message exchanging in a MAP [30]. They standarize the structure of an Agent Communication Language (ACL) message to ensure interoperability and also Content Language (CL) specifications for representing the content of the ACL messages. The use of standard specifications is vital in order to allow interoperability between heterogeneous agents which could compose an open system, as well as to define standard ontologies for accessing the MAP services.

Resource Description Framework (RDF) is a language for representing information about resources on the World Wide Web. By generalizing the concept of a ”Web resource”, RDF can also be used to represent information about things even when they cannot be directly retrieved from the Web [6]. RDF is based on the idea of identifying things using Web identifiers (called Uniform Resource Identifiers, or URIs), and describing resources in terms of simple properties and property values. The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object. The subject can be any resource, the predicate is a named property of the subject and the object denotes the value of this property. A set of such triples is called an RDF graph. RDF also provides an XML-based syntax (called RDF/XML [7]) for recording and exchanging these graphs. RDF is intended for situations in which this information needs to be processed by applications, rather than only being displayed to people. RDF provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning.

Due to the features of RDF and its widespread use in MAS [18, 24, 35, 36], an RDF-based framework for managing information has been designed for Magentix and has been integrated into it. It allows a Magentix agent to manage all of its information as RDF models (RDF graphs). Moreover, Magentix itself uses the framework for the messages exchanged, for representing the information that Magentix services manage, for interacting with the Magentix services and for storing MAP events.

The framework is based on offering an API to deal with RDF management. Of course, we did not implement an RDF support from scratch, the framework is designed as a wrapper for existing RDF management libraries and is aimed at simplifying the use of RDF inside a Magentix agent. There are some libraries that deal with RDF models. However, because of the Magentix features, i. e., the fact that it is implemented in C language and is focused on achieving high levels of efficiency, we have chosen the Redland libraries [8].

Redland is a set of free software C libraries that provide support for RDF. The authors of Redland claim that it is portable, fast and with no known memory leaks. It allows the manipulation of the RDF graph, triples, URIs and Literals. It can be implemented efficiently in C, providing memory storage with many databases (Berkeley DB, MySQL, etc.). We use the RDF/XML syntax to serialize the RDF graphs, but Redland also support other syntaxes, such as N-Triples or Turtle Terse RDF Triple Language. Queries can be carried out with SPARQL or RDQL.

One of the functionalities of Magentix where the RDF has been used in it is to represent messages. Agents and services use message sending to communicate with each other as we said in section 3.1. FIPA defines the structure of an Agent Communication Language (ACL) and also defines the use of RDF to represent the message content [31]. Message header and message content in Magentix are represented as RDF models serialized as XML. Some MAPs use this kind of serialization to represent the message content only (such as Jade), just as FIPA proposes. We provide Magentix with RDF to represent the whole message. Therefore, only one parser is needed and this simplifies the parsing and serializing process of a message.

As far as we are concerned, representing the FIPA-ACL using RDF should be standard, but currently it is not, so interoperability with other FIPA-compliant MAPs is compromised. A simple gateway that directly translates both representations can be added to solve this problem. Figure 2 shows an example of a Magentix message. It is an RDF graph in which resources are drawn as ellipses and literals are drawn as squares. As can be observed, all of the FIPA-ACL fields are mapped as RDF properties describing a message resource. The content of the message can also be seen as an RDF sub-graph inside the main RDF graph representing the message. Therefore, any information that a Magentix agent has as an RDF graph, can be added or retrieved directly from a message.

Regarding the representation of information about Magentix services, an ontology for interacting with them has been defined using Web Ontology Language (OWL) [5]. The ontology mainly focuses on describing the resources that the services manage (hosts, agents, services, organizational units, etc.). Therefore, all of the information is treated, without taking into account implementation concerns, so that a change in the implementation does not have any effect on the way the services treat the information. What is more, they can store all of its information in a direct and simper fashion on a database.

In order to achieve rich and flexible interactions between agents and Magentix services, the ontology also includes actions that can be requested by a Magentix service (creation of a new agent to the *AMS*, registering a service to the *DF*, creation of a new organizational unit to the *OUM*, etc.). Therefore, any Magentix agent that knows the ontology can interact with Magentix services and also manage all of the related knowledge using the framework provided.

4.5. Security Model

The Magentix MAP has a security model [48, 47], which is based on both the Kerberos protocol and the Linux OS access control. This model provides Magentix with authentication, integrity and confidentiality. By means of this model each agent has an identity which it can prove to the rest of the agents and services in a running Magentix MAP.

Magentix agents can have three identity types:

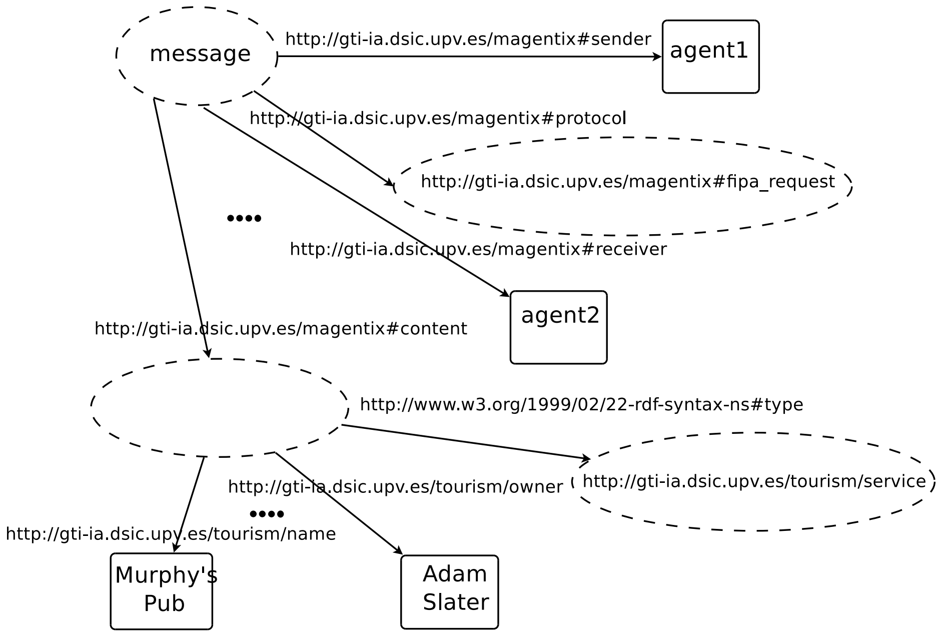
message
http://gti-ia.dsic.upv.es/magentix#sender
agent1
http://gti-ia.dsic.upv.es/magentix#protocol
http://gti-ia.dsic.upv.es/magentix#fipa_request
••••
http://gti-ia.dsic.upv.es/magentix#receiver
agent2
http://gti-ia.dsic.upv.es/magentix#content
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://gti-ia.dsic.upv.es/tourism/owner
http://gti-ia.dsic.upv.es/tourism/service
••••
http://gti-ia.dsic.upv.es/tourism/name
Murphy's Pub
Adam Slater

Fig. 2. Magentix Message represented in RDF

- **Agent** identity. Its identity as an agent. This identity is created by the AMS when the agent is created.
- **User** identity. The identity of its owner, i.e, the identity of the user that created the agent.
- **Unit** identity. The identity of each unit that the agent is in.

An agent always has at least its **Agent** identity and its owner's **User** identity. Therefore, a Magentix agent is provided with more than one identity, so a way of letting the Magentix communication module know which Kerberos credentials it has to use when sending a message is needed. This is done with a new field in the message header. If this field is in the message header of a message to be sent, the communication module tries to use the identity chosen; otherwise the corresponding agent identity is used. If the Kerberos credentials associated to the identity that the agent is requesting are not available, and the agent is trying to use an identity that it does not own for instance, the sending of the message fails.

Magentix services are based on information replication in each host. In order to check the integrity of this information and protect it from being accessible to non-authorized users, service communication needs to be secured. In order to do so, the administrator creates a *principal* (the *principal* is the unique name of a user or service allowed to authenticate using Kerberos) for each service with a random key that is saved by default in /etc/krb5.keytab. This file is secured using Linux OS access control and can only be accessed by the *root* user, so Magentix services have to run as *root* privileges.

When a service requires communication with another service, a security context is established as a client with the *principal* of the MAP administrator and as a server with the *principal* of the destination service. Using this security context the information sent is encrypted and a message integrity code is calculated. Therefore, the client is sure that the destination service is the service expected. Moreover, the destination service knows that it is being contacted by a service with the administrator's identity, so the destination service will serve all of the requests it receives. Thus, only MAP services can exchange information with each other.

Securing agent communication is similar to securing service communication, but agents use the identity that the `ams` agent has created for them when creating a security context to allow a secure interaction with each other.

In order to make efficient use of security contexts, a context cache has been added to each agent. This cache contains the corresponding security context associated with a destination agent. This cache is not related to the connections cache, so that, when a connection with an agent is closed, the associated security context is not lost.

5. User Agents

Agents in Magentix are represented as Linux processes. Internally, every agent is composed of Linux threads: a single thread for executing the agent tasks (main thread), a thread for sending messages (sender thread) and a thread for receiving messages (receiver thread). The *ams* agent manages the creation and deletion of the user agents launched on the same host. The GAT is shared between the *ams* agent and these user agents, so accessing the physical addresses of any agent of the MAP is fast and does not become a bottleneck. Agents have free read access to the GAT, thus, searching for the address of any agent registered in the MAP is efficient.

Magentix provides a template for developing agents written in C++. We provide different methods to manage the agent execution life cycle as well as the message sending and reception. Furthermore, agent developers can extend this model to include other requirements. Interaction with services is easily carried out by means of a specific API. Interactions among agents are focused on conversations. An agent can be interacting with several agents or services at any time. Each interaction between two agents can be represented as a pattern of communication where some messages are exchanged between the participants. These patterns can be predefined or not, but there is an initial message and a final message. The entire amount of messages exchanged between two participants represents a conversation. Magentix provides two functionalities for managing conversations: mailboxes and conversation managers.

5.1. Mailboxes

Mailboxes are used to improve the management of incoming messages from any agent. An agent is able to interact simultaneously with several agents. In these scenarios the possibility of distributing the incoming messages in different message queues, depending on the conversation that belongs each message, becomes interesting. By default every agent has a unique Mailbox called *DEFAULT_MAILBOX*, which receives every message addressed to the agent.

Magentix allows agent developer to create new Mailboxes and later, associate a conversation identifier to them. Then, when a message with this conversation identifier (represented as the *conversation_id* field of the message) is received, this message is routed to the corresponding Mailbox. This functionality allows messages to be filtered and split according to this field, so that, agent developers can easily distribute the different conversations which an agent is involved in among different Mailboxes. A Mailbox is not restricted to receive only the messages of a specific conversation identifier since we consider the possibility of associating several identifiers to the same Mailbox. The basic functionality an agent developer needs to bear in mind when working with Mailboxes is creating new Mailboxes and then, associating them to conversation identifiers. When an agent checks the message incoming queue, it specifies which Mailbox it wants to check. We can see in figure 3 an image of the internal structure of a Magentix agent.

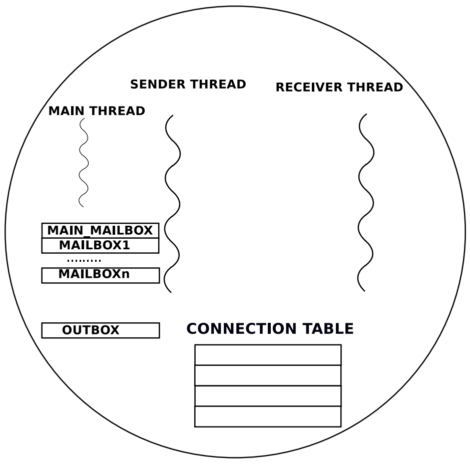

Fig. 3. Magentix Agent

5.2. Conversation Manager

Interactions between Magentix agents are focused on conversations. Thus, it is important for us not only to the searching and sending of messages to other agents but also to easily reproduce typical conversation patterns that can appear in a big variety of scenarios. An agent is able to simultaneously communicate with several agents. Every interaction between a pair of agents very often requires the exchanging of more than one message. Moreover, message exchange patterns are usually repeated in several interactions between agents, i.e. to access some service, to request information, to send proposals to different agents, etc. Thus, defining communication patterns to specify which messages exchanges are allowed for a specific interaction proves to be an interesting and useful feature for agent developers.

FIPA defines standard interaction protocol specifications that agents can use in their conversation with other agents ([32]). These specifications deal with pre-agreed message exchange protocols for ACL messages. Magentix provides support for executing these protocols defined by FIPA, therefore, agent developers can easily reproduce these interaction scenarios without needing to consider the sequence of exchanged messages, the possible failures in the execution of the protocol and so on. Agent developer only has to specify what to do when some of the deterministic events of the protocol take place and the protocol will automatically be checked and executed by Magentix.

Interaction protocols are defined by FIPA using UML-diagrams. In figure 4 we can see the protocol FIPA-request as an example. In these protocols there are two roles, initiator and participant, which exchange some possible message sequences. We translate this representation to Magentix as finite state machines. Each interaction protocol has a finite state machine associated to each possible role of the protocol. In figure 5 we can see the FIPA-request protocol for the initiator role. Each finite state machine has these properties:

- A *not create* initial state. This state is the first of every protocol.
- Transitions which allow the execution of the protocol depending on the messages received (represented as performatives such as refuse or agree) or λ-transitions, which take the protocol execution forward to the next state.
- Intermediate states for representing the intermediate steps of the protocol execution.
- A *delete* state. This is the last one of every protocol.

In order to process these interaction protocols we define a conversation manager. A conversation manager is an internal entity within Magentix agents, which has one or more interaction protocols associated to it. When an agent is using one of these protocols in its conversations with other agents, its conversation manager is in charge of automatically managing it and ensuring the correct execution of the protocol, executing each step and transition of the protocol. Several conversation managers can be assigned to a single agent, each one in charge of the management of different interaction protocols. This decision depends on the agent developer, which can run more conversation managers or

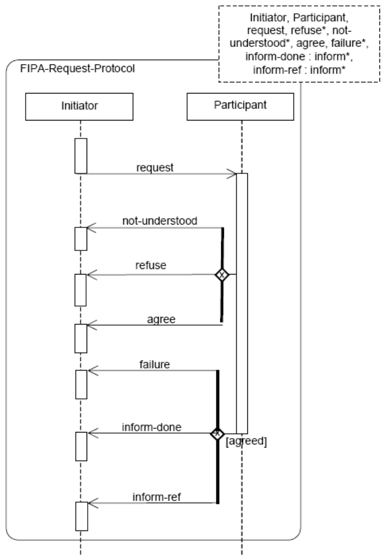
Initiator, Participant, request, refuse*, not-understood*, agree, failure*, inform-done : inform*, inform-ref : inform*
FIPA-Request-Protocol
Initiator
Participant
request
not-understood
refuse
agree
failure
inform-done
[agreed]
inform-ref

Fig. 4. FIPA-request Interaction Protocol

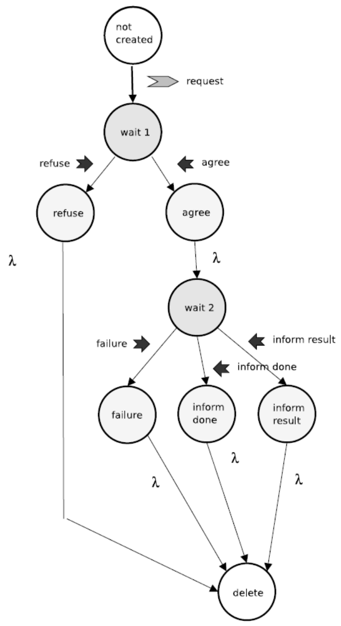

Fig. 5. Finite State Machine for FIPA-request in Magentix

stop them according to its needs. The conversation manager is an abstraction that hides the basic concepts of the conversations (makes sure the message is exchanged, mailbox management, etc.) from the agent developer, which only has to specify what to do in each step of the protocol, easily allowing the concurrent execution and management of several conversations. We are now working to extend the conversation management funcionalities. We especially want to facilitate the specification of any protocol interaction that agent developer could require, apart from that predefined by FIPA.

6. The Tourism Service Application

In this section, we present a real application developed in Magentix which uses some of the features provided. In order to test the performance of a MAP focused on large systems, we require examples aimed to be large-scale which are so real as possible. The Tourism Service application [39] is a MAS that allows users to find information about places of interest in a city according to their preferences (restaurants, movie theaters, museums, theaters, and other places of general interest such as monuments, churches, beaches, parks, etc.), by using their mobile phone or PDA. Once a specific place has been selected, the tourist can make a reservation at a restaurant, buy tickets for a film, etc. Our research group has been working with a partnership developing MAS-based recommender systems for tourists.

There are four different agent types in the application. A SightAgent manages all of the information related to the features and activities for a specific place of interest in the city. A UserAgent allows tourists to interact with the system by means of a GUI on their mobile devices. A BrokerAgent mediates between UserAgents and SightAgents. It also manages updated information about the SightAgents registered on it. Finally, a PlanAgent manages all of the planning processes in the system. The application offers search, reservation, planning, and registration services. The Search service is offered by the BrokerAgent and can be requested by a UserAgent. The result of the invocation of this service is a list of descriptions of places that match user preferences. The Reserve service is offered by a SightAgent and can be requested by a UserAgent. The result of this service is the confirmation of a successful reservation or an error message. The ”Plan a Specific Day” service is provided by the PlanAgent and can be requested by a UserAgent. The result of this service is a plan consisting of a list of places or activities.

We have implemented this application using the Magentix MAP with RDF support. The implemented ontology is represented in RDF and gives detailed descriptions of tourist places, information about scheduling, etc; For example, information about restaurants, represent issues related to menus, cuisine, ingredients, etc.

UserAgents can be implemented as Magentix agents in the MAP or by means of an interface that is implemented using the J2ME (Java 2 Micro Edition) specification. In the latter case, UserAgents have to make HTTP requests to a GatewayAgent, which acts as a gateway between UserAgents and the rest of the system. This GatewayAgent is implemented as a Magentix agent, which includes a micro-http server. This mechanism allows the interaction between Magentix agents and external agents.

7. Large Scale Evaluation of the Messaging Service

In this section, we present different experiments in order to evaluate the messaging service of Magentix, based on the application presented in Section 6. As we stated in Section 2, this service is crucial when developing systems with large agent populations with high message traffic. In [16], we presented a testbed for MAP performance evaluation. These tests focused on evaluating different parameters of the MAP in one and two hosts: the message traffic, the message size, the registered services, the searched services, the CPU consumption of the threads, the memory consumption, the network traffic, etc. According to these tests, the main bottleneck of a MAP performance is related to the messaging service. These conclusions have also been confirmed by other authors, who claim that other parameters such as the CPU cycles do not reach saturations in large-scale environments [28]. Based on these conclusions, in [49] we presented a set of large-scale benchmarks to test the messaging service. The experiments shown here are based on these benchmarks and are adapted to the Tourism Service Application presented in Section 6.

We compare Magentix against the performance of Jade, which is a well-known MAP and is more scalable than other MAPs as we stated in Section 2. Since the initial implementation of the Tourism Service Application was in Jade [38], we can determine the performance of the messaging service of both MAPs simulating different scenarios in this domain. We used 20 PCs Intel(R) Core(TM) 2 Duo CPU @ 2.60GHz, 2GB RAM, Ubuntu 10.10 and Linux Kernel 2.6.35. The computers were connected to each other via a 100Mb Ethernet hub.

The first experiment is aimed at testing the MAPs performance when both the number of agents and the message traffic increase. This experiment measures the capability of the MAP when messages are sent to different agents. As an example, this situation can occur when a BrokerAgent requests different SightAgents. We simulate this scenario by launching several groups of BrokerAgents and SightAgents. The objective of each BrokerAgent is to send a message to the first SightAgent on its list, which sends back the same message. After that, each BrokerAgent sends a message to its corresponding SightAgent placed in the next host and waits for the response. This experiment measures the time elapsed between when the first message is sent by the first BrokerAgent and when the last message is received by the last BrokerAgent. The experiment started with 100 agents in the system, increasing to 1000. The number of messages sent by each BrokerAgent was specified at 1000.

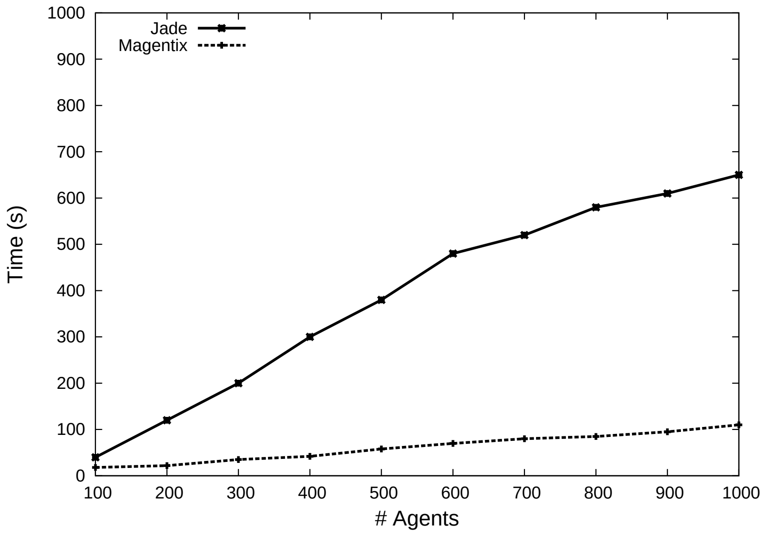

Fig. 6. Experiment 1: population and traffic increase

Figure 6 shows the time required for the two MAPs. The figure shows that there is a performance degradation as the number of agents and the message traffic increase. However, Magentix performance degrades less than Jade performance. As an example, it can be observed that the elapsed time in Magentix when the system is composed of 1000 agents is less than the elapsed time in Jade when the system is composed of 200 agents.

Another typical scenario is the massive amount of message-sending to a specific agent. The second experiment measures the ability of the MAPs when a lot of agents send messages to a single one. This specific agent could become a bottleneck in the system when multiple messages are addressed to it. This scenario appears, for example, when UserAgents are requesting the same BrokerAgent to retrieve information. The BrokerAgent has to serve every received request. As the number of incoming requests increases, the time for processing these requests may also increase. In order to simulate this, a single BrokerAgent agent and several UserAgents were launched. The goal of each UserAgent was to send messages to the BrokerAgent. The elapsed time between when the BrokerAgent received the first message and when it answered all the messages is shown in Figure 7. In this experiment, we increased the number of UserAgents up to 100, distributed among all the hosts. Each UserAgent sent 10000 messages.

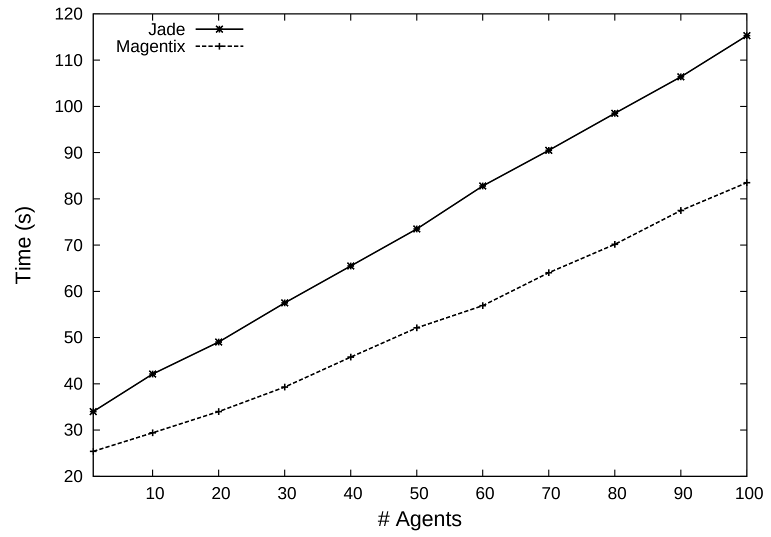

Fig. 7. Experiment 2: massive sending to an agent

It can be observed that the elapsed time increases in both MAPs as the number of requests increases. However, as in the first experiment, the performance degradation is less in Magentix. The time difference between the two

MAPs gradually increases as the number of agents increases. Therefore, Magentix is also more scalable and efficient than Jade in this scenario. Note that in this scenario the receiver agent is not changed during the entire experiment.

The third experiment complements the second one. The distribution of agents in this experiment was similar. However, there were the same number of BrokerAgents as UserAgents. In this experiment, several BrokerAgents were placed in the same host and each UserAgent communicated with its corresponding BrokerAgent. The results obtained are shown in Figure 8. It can be observed that the results for Jade are similar to the results for the second experiment. This is due to the way that Jade implements communication among all the MAP hosts. Therefore, the bottleneck is caused by the message transport system and not by the way the message queue is managed by the agent itself. In contrast, the performance in Magentix in the third experiment is slightly better than in the second one.

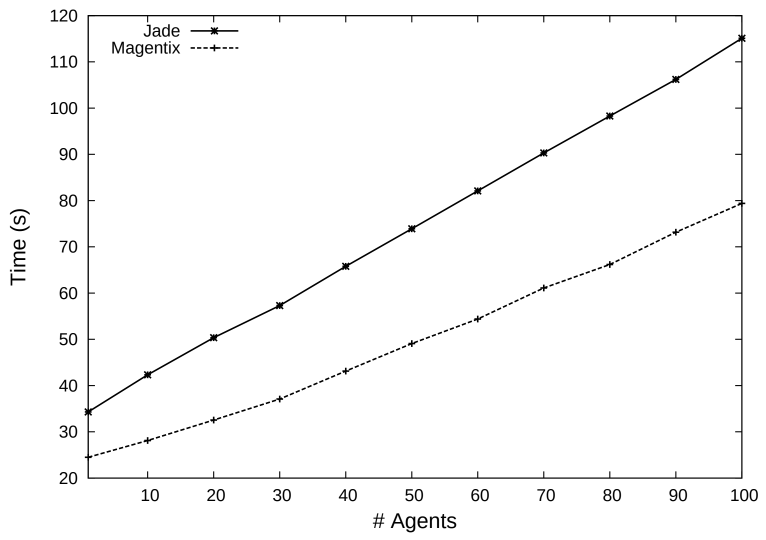

Fig. 8. Experiment 3: host massive sending

The fourth experiment checks the limits of the MAPs. This experiment provides a different perspective from the previous experiments in which the receiver agents are predefined. This may give rise to different bottlenecks, showing another typical scenario in real systems, in which some agents may be more requested than others. In order to simulate this, several BrokerAgents were placed in 10 hosts of the MAP and several UserAgents were placed in the other 10 hosts. Each UserAgent had to send 1000 messages to a non-predefined BrokerAgent. Thus, the specific BrokerAgent was randomly selected before sending each message. This caused some BrokerAgents to be more overloaded than others. Furthermore, in this experiment, the number of agents was increased to 2000, in order to overload the MAPs.

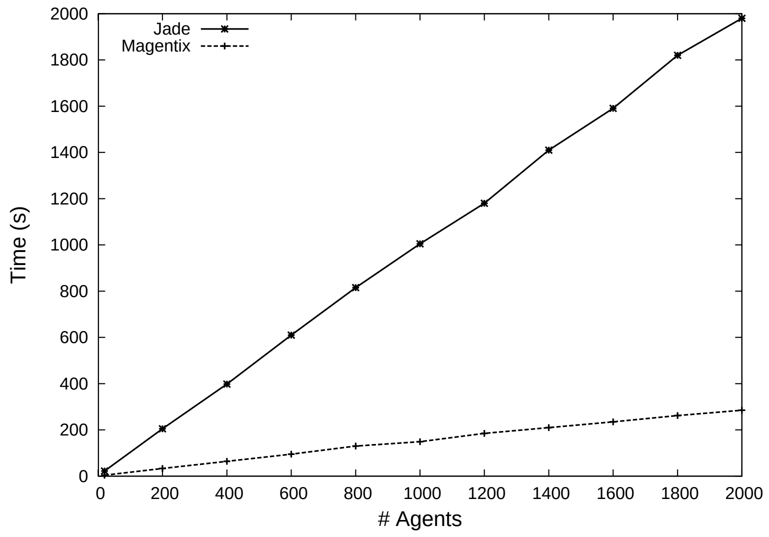

Fig. 9. Experiment 4: random requests

It can be observed in Figure 9 that Magentix offers better performance than Jade, and the differences increase according to the increase in the traffic. The figure also shows that the two MAPs present higher response times with respect to the first experiment, in which the traffic was equally distributed among all the BrokerAgents. This is due to the fact that, in this forth experiment, message load is not spread over all of the receiver agents launched. Since the BrokerAgent in each message sending is selected randomly, there may be BrokerAgents that have to serve a lot of messages while others are idle. Therefore, as the second experiment indicates, Jade performs quite badly when there is an agent that is receiving a lot of messages. As a result, performance differences with respect to the first experiment are much higher in Jade than in Magentix.

From the results provided in these tests, we can conclude that Magentix improves the efficiency and scalability of the messaging service provided by Jade, which is the most commonly used MAP and that it is more scalable than other MAPs. In these tests, we have simulated four typical scenarios in order to determine the efficiency and scalability in the Magentix and the Jade MAPs. These tests represent critical situations so that we can see the degree of performance improvement achieved more clearly. Although we scale up to 20 hosts in these tests, the conclusions obtained can be extended to at least to 100 hosts according to the results shown in [49].

8. Conclusions

The next generation of technologies aims to provide features such as distribution, interoperability, scalability, organizations, service-oriented, open, geographically dispersed, and so on. MASs can contribute to these environments by evolving new applications that will become more autonomous and social from the point of view of the MAS field.

MAPs have traditionally been used as a support framework to facilitate the development of these kinds of systems. A lot of MAPs have been developed in the last few years; however, unfortunately, very few real MAS-based applications have appeared, probably due to the lack of suitability of the support frameworks which did not fulfill all of the requirements. In order to support the new generation of systems (in line with the latest trends in rapidly expanding technologies), new MAP designs should focus on being interoperable, scalable, and large-scale as just some of their key features.

In this paper, we have presented the Magentix MAP. Since its design is closer to the OS level, it ensures that the MAP is efficient, especially when running large systems. Basic services such as an agent directory service, a service directory service, and a messaging service are provided by Magentix. We have implemented and tested the performance of this MAP. Magentix also provides a group-oriented communication mechanism. This mechanism allows communication between individual agents as well as interaction among groups of agents. When considering large systems, security concerns become an important issue and a necessary feature when these systems become open. Magentix has a security model that is based on the Kerberos protocol and Linux OS access control which provides authentication, integrity, and confidentiality. In order to achieve interoperable systems, we represented the information using RDF. This framework has been widely used in MAS for different purposes. Magentix represents messages to be exchanged in RDF so that agents can easily manage the information that is sent and received. Ontologies defined in OWL have also used to interact with services.

Using a tourism service application, we have shown how Magentix can be used as a support framework to develop MAS-based applications. The messaging service evaluation shown in this paper demonstrates that a MAP design that uses the OS services provides greater efficiency and scalability than other high-performance middleware-based MAPs such as Jade.

With the features provided by Magentix we can establish the next objective of the project: to provide Magentix with support for open MAS. We are working on the development of an http-based gateway at MAP level, in order to allow the interaction between Magentix agents and agents developed in other MAPs. Virtual organizations where agents dynamically enter and exit the system and form groups could also be created in Magentix.

Acknowledgments. This work has been partially supported by CONSOLIDER-INGENIO 2010 under grant CSD2007-00022, and projects TIN2011-27652-C03-01 and TIN2008-

04446. Juan M. Alberola has received a grant from Ministerio de Ciencia e Innovación de España (AP2007-00289).

References

1. Fipa-os. http://fipa-os.sourceforge.net
2. FIPA (The Foundation for Intelligent Physical Agents). http://www.fipa.org/
3. Jack. http://www.agent-software.com
4. Madkit. http://www.madkit.org
5. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/
6. RDF. http://www.w3.org/TR/rdf-primer/
7. RDF/XML Syntax Specification. http://www.w3.org/TR/rdf-syntax-grammar/
8. Redland RDF Libraries. http://librdf.org
9. Safeguard. http://www.ist-safeguard.org/
10. Standard for information technology - portable operating system interface (POSIX)
11. Tryllian agent development kit (adk). http://www.tryllian.com
12. Zeus agent toolkit. http://labs.bt.com/projects/agents/zeus/
13. SACI - simple agent communication infrastructure. http://www.lti.pcs.usp.br/saci/ (2009)
14. Alberola, J.M., Mulet, L., Such, J.M., Garcia-Fornes, A., Espinosa, A., Botti, V.: Operating system aware multiagent platform design. In: Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS-2007). pp. 658–667 (2007)
15. Alberola, J.M., Such, J.M., Espinosa, A., Botti, V., Garcia-Fornes, A.: Scalable and efficient multiagent platform closer to the operating system. Artificial Intelligence Research and Development 184, 7–15 (2008)
16. Alberola, J.M., Such, J.M., Garcia-Fornes, A., Espinosa, A., Botti, V.: A performance evaluation of three multiagent platforms. In: Artificial Intelligence Review, Volume 34, Number 2. pp. 145–176 (2010)
17. Batouma, N., Sourrouille, J.L.: Dynamic adaption of resource aware distributed applications. In: International journal of grid and distributed computing. vol. 4, pp. 25–42 (2011)
18. Bauwens, B.: Xml-based agent communication: Vpn provisioning as a case study. In: XML Europe'99 (1999)
19. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: Jade a white paper. EXP 3, 6–19 (2003)
20. Bitting, E., C.J.G.A.: Multiagent system development kit: An evaluation. In: Proceedings of Communication Networks and Services Research Conference, May 15-16, pp. 80-92, Moncton, New Brunswick, Canada, 2003
21. Bădică, C., Budimac, Z., Burkhard, H.D., Ivanovic, M.: Software Agents: Languages, Tools, Platforms. Computer Science and Information Systems 8(2), 255–298 (2011)
22. Burbeck, K., Garpe, D., Nadjm-Tehrani, S.: Scale-up and performance studies of three agent platforms. In: IPCCC 2004 (2004)
23. Camacho, D., Aler, R., Castro, C., Molina, J.M.: Performance evaluation of zeus, jade, and skeletonagent frameworks. In: IEEE International Conference on Systems, Man and Cybernetics, 2002 (2002)
24. Cenk, R., Dikenelli, O., Seylan, I., Gürcan, Ö.: An infrastructure for the semantic integration of fipa compliant agent platforms. In: AAMAS. pp. 1316–1317 (2004)
25. Chmiel, K., T.D.G.M.K.P.: Testing the efficency of jade agent platform. In: Proceedings of the ISPDC/HeteroPar'04, 49-56 (2004)

26. Cortese, E., F.Quarta, Vitaglione, G.: Scalability and performance of jade message transport system. EXP 3, 52–65 (2003)
27. E. Argente, A. Gilet, S.V.V.J., Botti, V.: Survey of mas methods and platforms focusing on organizational concepts. In: Recent advances in Artificial Intelligence Research and Development. vol. 113, pp. 309–316. IOS Press (2004)
28. Fernández, V., Grimaldo, F., Lozano, M., Orduña, J.M.: Evaluating jason for distributed crowd simulations. In: ICAART (2). pp. 206–211 (2010)
29. FIPA: FIPA Abstract Architecture Specification. FIPA (2001), http://www.fipa.org/specs/fipa00001/
30. FIPA: FIPA ACL Message Structure Specification. FIPA (2001), http://www.fipa.org/specs/fipa00061/
31. FIPA: FIPA RDF Content Language Specification. FIPA (2001), http://www.fipa.org/specs/fipa00011/
32. FIPA: FIPA Interaction Protocol Library Specification. FIPA (2003), http://www.fipa.org/specs/fipa00025/
33. Giang, N.T., Tung, D.T.: Agent platform evaluation and comparison (2002)
34. Hirsch, B., Konnerth, T., Heßler, A.: Merging agents and services — the JIAC agent platform. In: Multi-Agent Programming: Languages, Tools and Applications, pp. 159–185. Springer (2009)
35. Huynh, D., Karger, D.R., Quan, D.: Haystack: A platform for creating, organizing and visualizing information using rdf. In: Eleventh World Wide Web Conference Semantic Web Workshop (2002)
36. Laclavik, M., Balogh, Z., Gatial, E., Hluchy, L.: Agent architecture based on semantic knowledge model. In: 5th annual conference. VSB-Technick. pp. 288–291 (2006)
37. Lee, L.C., Ndumu, D.T., Wilde, P.D.: The stability, scalability and performance of multi-agent systems. BT Technology Journal 16, 94–103 (1998)
38. Lopez, J.S., Bustos, F.A., Julian, V., Rebollo, M.: Developing a Multiagent Recommender System: A Case Study in Tourism Industry. International Transactions on Systems Science and Applications 4(3), 206–212 (2008)
39. Lopez, J.S., Bustos, F.A., Inglada, V.J.: Tourism services using agent technology: A multiagent approach. INFOCOMP - Journal of Computer Science - Special Edition pp. 51–57 (2007)
40. Lynch, S.: Using meta-agents to build mas platforms and middleware. In: International Conference on Agents and Artificial Intelligence (ICAART) (2011)
41. Mulet, L., Such, J.M., Alberola, J.M.: Performance evaluation of open-source multiagent platforms. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06). pp. 1107–1109. Association for Computing Machinery, Inc. (ACM Press) (2006)
42. Nodine, M.H., Unruh, A.: Facilitating open communication in agent systems: the infosleuth infrastructure (1997)
43. Omicini, A., Rimassa, G.: Towards seamless agent middleware. In: TAPOC 2004
44. Park, A.H., et al.: A flexible and scalable agent platform for multi-agent systems. In: Proceedings of WASET Bangkok (2007) (2007)
45. Pesovic, D., Vidakovic, M., Ivanovic, M., Budimac, Z., Vidakovic, J.: Usage of agents in document management. Computer Science and Information Systems 8(1), 193–210 (2011)
46. Shakshuki, E.: A methodology for evaluating agent toolkits. In: ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I. pp. 391–396. IEEE Computer Society, Washington, DC, USA (2005)

47. Such, J.M., Alberola, J.M., Espinosa, A., Garca-Fornes, A.: A Group-oriented Secure Multiagent Platform. Software: Practice and Experience 41(11), 1289–1302 (2011)
48. Such, J.M., Alberola, J.M., Garca-Fornes, A., Espinosa, A., Botti, V.: Kerberos-based secure multiagent platform. In: Sixth International Workshop on Programming Multi-Agent Systems (ProMAS'08). pp. 173–186 (2008)
49. Such, J.M., Alberola, J.M., Mulet, L., Espinosa, A., Garcia-Fornes, A., Botti, V.: Large-scale multiagent platform benchmarks. In: LAnguages, methodologies and Development tools for multi-agent systemS (LADS 2007). Proceedings of the Multi-Agent Logics, Languages, and Organisations - Federated Workshops. pp. 192–204 (2007)
50. Such, J.M., Alberola, J.M., Mulet, L., Garcia-Fornes, A., Espinosa, A., Botti, V.: Hacia el diseo de plataformas multiagente cercanas al sistema operativo. In: International workwhop on practical applications on agents and multi-agent systems (2007)
51. Vrba, P.: Java-based agent platform evaluation. In: Proceedings of the HoloMAS 2003. pp. 47–58 (2003)

Juan M. Alberola is a PhD student at the Departament de Sistemes Informàtics i Computació of the Universitat Politècnica de València. His interest areas include agent organizations, adaptation, multiagent platforms, case-based-reasoning and electronic markets.

Jose M. Such is Lecturer in the School of Computing and Communications at Lancaster University (UK). He was previously research fellow at Universitat Politècnica de València (Spain), by which he was awarded a PhD in Computer Science in 2011. He is mostly interested in the following research topics: Privacy, Security, Trust, Reputation, Multi-agent Systems, and Artificial Intelligence.

Vicent Botti is Full Professor at the Universitat Politècnica de València (Spain) and head of the GTI-IA research group of the Departament de Sistemes Informàtics i Computació. He received his Ph.D. in Computer Science from the same university in 1990. His research interests are multi-agent systems, agreement technologies, and articial intelligence, where he has more than 200 refereed publications in international journals and conferences. Currently he is Vice-rector of the Universitat Politècnica de València.

Agustín Espinosa is Lecturer at the Departament de Sistemes Informàtics i Computació of the Universitat Politècnica de València and a researcher at the GTI-IA Research Group of the Universitat Politècnica de València. His research interests include multiagent systems, agent architectures, agent platforms, agent frameworks, and real-time agents. He received his Ph.D. in Computer Science from the Universitat Politècnica de València, Spain in 2003.

Ana García-Fornes is a Professor at the Departament de Sistemes Informàtics i Computació of the Universitat Politècnica de València. Her interest areas include: real-time artificial intelligence, real-time systems, development of multiagent infrastructures, tracing systems, operating systems based on agents, agent organizations, and negotiation strategies.

Received: October 29, 2011; Accepted: October 8, 2012.