Artif Intell Rev (2010) 34:145–176
DOI 10.1007/s10462-010-9167-9

A performance evaluation of three multiagent platforms

Juan M. Alberola · Jose M. Such · Ana Garcia-Fornes · Agustin Espinosa · Vicent Botti

Published online: 4 June 2010
© Springer Science+Business Media B.V. 2010

Abstract In the last few years, many researchers have focused on testing the performance of Multiagent Platforms. Results obtained show a lack of performance and scalability on current Multiagent Platforms, but the existing research does not tackle poor efficiency causes. This article is aimed not only at testing the performance of Multiagent Platforms but also the discovery of Multiagent Platform design decisions that can lead to these deficiencies. Therefore, we are able to understand to what extent the internal design of a Multiagent Platform affects its performance. The experiments performed are focused on the features involved in agent communication.

Keywords Multiagent platforms · Multiagent systems · Performance evaluation

This work has been partially supported by CONSOLIDER-INGENIO 2010 under grant CSD2007-00022, and projects TIN2008-04446 and PROMETEO/2008/051. Juan M. Alberola has received a grant of Ministerio de Ciencia e Innovación de España (AP2007-00289). Jose M. Such has received a grant of *Conselleria d'Empresa, Universitat i Ciència de la Generalitat Valenciana* (BFPI06/096).

J. M. Alberola (✉) · J. M. Such · A. Garcia-Fornes · A. Espinosa · V. Botti
Departament de Sistemes informàtics i Computació, Universitat Politècnica de València,
Camí de Vera s/n, 46022 Valencia, Spain
e-mail: jalberola@dsic.upv.es

J. M. Such
e-mail: jsuch@dsic.upv.es

A. Garcia-Fornes
e-mail: agarcia@dsic.upv.es

A. Espinosa
e-mail: aespinos@dsic.upv.es

V. Botti
e-mail: vbotti@dsic.upv.es

1 Introduction

Multiagent Systems (MASs) development has increased in the last few years, in the fields of both research and industry. MAS developers need software that helps them in the developing process. Multiagent Platforms (MAPs) provide some tools that improve the development and implementation of MASs. Due to the large number of existing MAPs, the choosing of a suitable one to develop a MAS becomes a difficult task for MAS developers.

As stated in Luck et al. (2005), the next generation of computing systems is likely to demand a large number of interacting components, be they services, agents or otherwise. Current tools work well with limited numbers of agents, but are generally not yet suitable for the development of large-scale (and efficient) agent systems, nor do they offer development, management or monitoring facilities able to deal with large amounts of information or tune the behavior of the system in such cases.

In the last few years, many researchers have focused on testing the performance of existing MAPs. Some studies like Mulet et al. (2006), Chmiel et al. (2004), Vrba (2003), Silva et al. (2000), Camacho et al. (2002), Burbeck et al. (2004), Cortese et al. (2003), Lee et al. (1998) focus on testing the performance of some MAPs; Bitting et al. (2003), Giang and Tung (2002), Shakshuki (2005) and Ricordel and Demazeau (2000) classify MAPs using criteria catalogues which focus on several features of MAPs: performance, security, availability, environment, development, etc.; finally, Omicini and Rimassa (2004) gives a brief evolution of MAPs.

Most of these studies highlight a lack of performance and scalability on current MAPs, but without providing reasons for such weaknesses. We want to perform an in-depth study of some MAPs in order to find out their weaknesses. As stated in Lee et al. (1998), performance of MAPs can be obtained by measuring the response time of the services they provide (e.g. time to send a message, to register a service, to search for a service, etc.), the resources they use (e.g. amount of memory, CPU time and network bandwidth needed) and the number of concurrent agents in the system. Moreover, scalability refers to how well the MAS performance adapts to increases in the size of the system Lee et al. (1998) (by increasing the number of the agents being executed and the number of requests they do for the services offered by the MAPs).

To achieve this goal, we have studied some well-known MAPs in order to understand to what extent the internal design of a MAP influences its performance. Among all of the current MAPs, only those whose source code is available are feasible for us, since part of our study requires the inspection of specific parts of the code.

Three of these MAPs have been selected: Jade (Bellifemine et al. 2003), MadKit (Gutknecht and Ferber 2000) and AgentScape (Brazier et al. 2002). These three MAPs are representative of different design principles and different programming models.

The behavior of a MAP is based on the services it offers. Indeed, the overall MAP performance is likely to be affected by the way in which services are designed. This article includes the definition of specific experiments to measure performance in some significant scenarios, the results obtained, and the explanation of those results in terms of MAP design principles. The experiments are focused specifically on the messaging and service directory[1] services. Other experiments regarding MAP memory usage, network occupancy, and CPU time usage of each MAP thread are discussed. Thus, we present an in-depth study of the three selected MAPs, from internal structure design and bottlenecks to performance.

[1] For the sake of clarity we will denote only *directory service* instead of *service directory service* in the rest of the article.

According to Wooldridge and Jennings (1995), an agent is defined by its flexibility, which implies that an agent is: reactive, an agent must answer to its environment; proactive, an agent has to be able to try to fulfill its own plans or objectives; and social, an agent has to be able to communicate with other agents by means of some kind of language. Moreover, as stated in Wooldridge (2002), a MAS consists of a number of agents that interact with one-another. Since sociability is a key feature of agents, many messages are exchanged in a MAS, which means that the messaging service is a key service in every MAP and its performance may determine MAS application efficiency. Furthermore, there are some MAP services that are implemented using messaging service, e.g. directory service in Jade.

There are other services that may have an impact on the overall performance of a MAP such as directory, mobility, security, logging, and so on. However, only the directory service is offered by all three of the MAPs selected. Therefore, an evaluation comparing the other services mentioned among the three MAPs selected is not possible.

Another reason to choose the messaging and directory services is the fact that both of them are defined by the Foundation for Intelligent Physical Agents (FIPA 2000) standard, currently the most widely accepted.

The rest of the article is organized as follows. Section 2 details some previous related works. Section 3 gives an overview of current MAPs. Section 4 gives a brief description of each selected MAP. Section 5 describes the experiments performed. Finally, in Sect. 6 there are some concluding remarks.

2 Related 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. Therefore, these works usually show the performance of several MAPs when running MASs which are composed by sender and receiver agents. Vrba (2003) presents an evaluation of the messaging service performance of four MAPs. From the tests presented in this paper the authors conclude that Jade provides the most efficient messaging service (among the four tested MAPs) and ZEUS(Collis et al. 1998) the worst one (among the four tested MAPs). However, the design reasons that cause 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 the better than the other three MAPs tested. Burbeck et al. (2004) test the messaging service performance of three MAPs. They claim that Jade performs better than the others because is built on Java RMI[2] but give no proofs confirming this claim. As far as we are concerned, these conclusions do not provide any clue for MAP developers to improve MAP designs. Moreover, these experiments scale up only to 100 pairs of agents. A deeper study is required in order to assess MAP performance and to what extent design decisions influence MAP performance. Furthermore, other services apart from the messaging service should also be tested.

Some other works test the performance of other services but only for a single MAP. Most of these works test the Jade MAP which seems to be the most used one. Chmiel et al. (2004) test the Jade messaging, agent creation and migration services. The tests they perform related to the messaging service only scale up to 8 agent pairs. Cortese et al. (2003) test the scalability and performance of the Jade messaging service. As in the works cited in the previous paragraph, conclusions are not related to any design decision. Therefore, these conclusions

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

can allow MAS developers to check if Jade fulfills their requirements when designing a MAS, but they do not suggest any design decision for MAP developers.

There are also other works that are not focused on testing MAP performance but the performance of a specific MAS running on top of a MAP. Camacho et al. (2002) shows the performance of MAPs when a MAS composed by a several web agents is launched. This MAS provides documents requested by a user agent. In this sense, the authors measure the number of requested documents per time. Therefore, the conclusions are only valid for this MAS. Lee et al. (1998) presents a MAS in which agents coordinate themselves for carrying out tasks. They evaluate how the topological relations between agents affects the number of CPU times needed to accomplish these tasks.

Finally, other studies are focused on detailing the functional properties of MAP. In Bitting et al. (2003) four MAPs are compared according several criteria: implementation languages, tools provided, agent deliberation capabilities and so on. Shakshuki (2005) presents a methodology to evaluate MAPs based on several criteria such as availability, environment, development, and so on. A similar work is carried out in Giang and Tung (2002). Omicini and Rimassa (2004) gives a brief evolution of MAPs. These works provide ratings of properties provided by MAPs in order to help users to choose the MAP according their needs. Our work is not only intended to be useful for MAP users (MAS developers) but also for MAP developers.

3 Overview of current multiagent platforms

The main purpose of a MAP is to provide a framework for the development and execution of agent systems. Nowadays, a MAP is always implemented as a middleware between the operating system and the agents. As MASs usually need to be executed in distributed environments, a MAP may consist of a set of hosts running agents. We can also see a MAP as a set of services for managing and developing applications based on MASs. For instance, a MAP must allow the creation and deletion of agents, as well as the management of their life cycle.

As well as offering services and facilities for designing and developing MAS based applications, MAPs provide some abstractions. Thus, these applications can be managed and designed without taking into account internal aspects of the MAP.

There are many MAPs developed by research groups around the world. The large number of MAPs available makes it difficult to select a specific one to create a MAS. This paper aims to analyze the current MAP deficiencies through the in-depth research of MAPs and their internal design. A full study of every MAP is not feasible due to the large number in existence. So, it is necessary to select some from among them to perform the tests.

Table 1 shows the results of a study of current MAPs taking into account some of the features they provide. These features were considered to choose the MAPs we test in this article, as shown later on in this section. The **Language** field shows the language in which each MAP is implemented, **O.S.** indicates whether a MAP is open-source or not, **FIPA** is set if the MAP is FIPA-compliant, **Sec.** points out whether security is taken into account, **Org.** shows if agent organizations are supported, and finally, **Comm** indicates what communication technology each MAP uses. A "$\surd$" mark indicates that a MAP has a feature. On the other hand, a "–" mark indicates that a MAP does not have a feature.

Most of these MAPs are usually implemented in Java and run on top of an operating system. Other languages like Python and C# are also used to develop MAPs. As most of the programming languages used when developing MAPs are interpreted, the execution of these

Table 1 Features provided by various multiagent platforms

Platform	Language	O.S.	FIPA	Sec.	Org.	Comm
3APL (Ten Hoeve 2003)	Java	√	√	–	–	RMI
AAP (Dale 2002)	April	√	√	–	–	ICM
ABLE (Bigus et al. 2002)	Java		√		–	RMI
ADK (Xu and Shatz 2003)	Java	–	–	√	–	
AgentDock (Jarvinen 2002)	Java	–	√			RMI
AgentScape (Brazier et al. 2002)	Java/Python	√	–	–	–	SunRPC
Aglets (Venners 1997)	Java	√	–	√	–	RMI
Ajanta (Tripathi et al. 2002)	Java	√	–	√	–	RMI
Ara (Peine and Stolpmann 1997)		√	–	√	–	RMI
CAPA (Duvigneau et al. 2003)	Java		√	–	–	
CapNet (Contreras et al. 2004)	C#	–	√	√	–	Several
Concordia (Ita 1997)	Java	–	–	√	–	RMI
Cougaar (Helsinger et al. 2004)	Java	√	–	√	–	RMI/Corba/http
CrossBow (Kusek et al. 2004)	Java				–	Proxy
Cybele (Inc 2004)	Java	√	–		–	
Dagents (Gray 1995)		√	–	√		RPC
Decaf (Graham et al. 2003)	Java	√	–		–	
Genie (Riekki et al. 2003)	Java	√	√	–	–	RMI
Grasshopper (Bäumer et al. 1999)	Java	–	√	√	–	RMI
Gypsy (Lugmayr 1999)	Java	√	–	√	–	RMI
Hive (Minar et al. 1999)	Java	√	–	√	–	RMI
Jade (Bellifemine et al. 2003)	Java	√	√	–	–	RMI/Corba/http
Jack (AOS Group 2008)	Java	–	–	–	√	tcp/ip
Jackal (Cost et al. 1998)	Java		–	–	√	tcp/ip
Jason (Bordini et al. 2007)	Java	√	–	–	√	Jade
Mage (Shi et al. 2004)	Java	√	√		–	RMI
MadKit (Gutknecht and Ferber 2000)	Java	√	–	–	√	Sockets
Sage (Ahmad et al. 2005)	Java	√	√	√	–	RMI
Semoa (Roth and Jalali-Sohi 2001)	Java	√	–	√	–	RMI
Soma (Bellavista et al. 1999)	Java	√	–	√		Corba
Spade (Escriva et al. 2006)	Python	√	√	√	√	Jabber
Spyse (Meyer 2004)	Python	√	√		–	
Voyager (Software 2007)	Java	–	–	√	–	RMI/Corba
Zeus (Collis et al. 1998)	Java	√	√	√	√	

MAPs requires an extra software layer, e.g. Java Virtual Machine for MAPs implemented in the Java programming language.

Another feature of MAPs is source code availability. Open source code is a code that is available to the general public with relaxed or non-existent intellectual property restrictions. However, closed source code is a code that is not available. Thus, source code availability is an important feature when the inspection of the source code of an application is required. Indeed, the source code availability plays a crucial role in order to assess to what extent implementation details affect performance.

FIPA specifications represent a collection of standards which are intended to promote the interoperability of heterogeneous agents and the services that they offer. Being interoperable with other MAPs is a desired feature, so FIPA compliance is important for us. As one can see in Table 1, there are several MAPs (over 40%) that are FIPA compliant.

Most current MAPs use Java Remote Method Invocation (Java RMI) to implement their communications. There are some exceptions, such as AgentScape (Brazier et al. 2002), which uses SunRPC, Spade (Escriva et al. 2006), which uses Jabber,[3] MadKit (Gutknecht and Ferber 2000), which uses raw Java TCP Sockets, and AAP (Dale 2002), which uses Interagent Communication Module[4] (ICM), to name a few. Java RMI, Java TCP Sockets and SunRPC technologies are further explained in Sect. 5.1.1.

Security issues are not usually taken into account on current MAPs. Only a few of them have their own security mechanisms for controlling agent execution on the MAP. Among them are Semoa (Roth and Jalali-Sohi 2001), which uses filters and agent grants on each node of the MAP, Cougaar (Helsinger et al. 2004), which utilizes authentication and encryption, Soma (Bellavista et al. 1999), which applies a policy to protect agents from each other, CapNet (Contreras et al. 2004), which has a policy manager and agent credentials, and so on.

Another important functionality that a MAP should offer is an organizational/relation model. Agent organizations allow agents to coordinate, structure and cooperate with each other. As stated in Argente et al. (2005) and Nwana (1994), agents need these for several reasons: to prevent anarchy or chaos, to fulfill global restrictions, to share and exchange knowledge, to control interdependences between agent actions, to improve efficiency, etc. This concept becomes increasingly important in complex systems in which hundreds of agents are running, with many interactions among them. As one can see in Table 1, only a few MAPs provide a minimal organizational/relation model. For example, MadKit (Gutknecht and Ferber 2000) offers an Agent/Group/Role model to implement relations among agents. Jackal (Cost et al. 1998) has a Conversation Manager that establishes rules in every agent communication and Zeus (Collis et al. 1998) provides a tool that helps users to define agent relations.

There are many other criteria for choosing a MAP such as the degree of support, documentation, tutorials, availability of software for different operating systems, support for development (editors, IDEs, libraries and APIs), discussion forums, and so on. We know that these other issues are also very important and should be taken into account. In this work we do not analyze these features.

After an analysis of several current MAPs has been carried out, three of them are chosen to be evaluated. All of the features described above have been taken into account in our selection. Jade, MadKit and AgentScape have been chosen. We need to study the source code of the MAPs to carry out an in-depth study of their internal design, so the three MAPs selected are Open Source. As stated in Bellifemine et al. (2008), Jade can arguably be considered the most popular MAP available today, and it is FIPA compliant. MadKit provides an organizational/relation model based on Agent, Group and Role concepts. AgentScape MAP is not only implemented in Java but also in Python, and according to the AgentScape developers, its approach to management is targeted to scalability and autonomy.

In order to evaluate the performance of the selected MAPs, an in-depth study of these MAPs is carried out in this article.

[3] http://www.jabberes.org/.

[4] http://www.nar.fujitsulabs.com/icm/.

4 Description of selected platforms

This section introduces the three selected MAPs. It only shows the basic concepts of each MAP. In Sect. 5, more specific features are explained in each experiment description, focusing on the issues of the internal design of each MAP that could affect the results obtained.

Java Agent DEvelopment Framework, (Bellifemine et al. 2003) (Jade) is a MAP that is entirely built in Java. It is made by Telecom Italia Lab and its code is open source. This MAP has become the most frequently used by MAS developers. There are some advantages of using Jade: it is FIPA compliant, and it offers a complete API that makes agent programming easier.

Jade architecture is based on the container concept. A container provides the environment and services required for executing agents. There is a special container called *Main Container*, which is the container launched by default when the MAP is issued. The Agent Management Service (AMS) and Directory Facilitator (DF) FIPA agents are located in this container. A single container is usually located per host in the MAP, but more than one container can stay in the same host.

Multi Agent Development KIT, (Gutknecht and Ferber 2000) (MadKit) is a non FIPA-compliant MAP. It is a project developed by LIRMM (Laboratorie d'Informatique de Robotique et de Microélectronique de Montpellier). The MAP is fully implemented in Java and is based on the agent-group-role (*AGR*) model.

The MadKit micro-kernel is a small, optimized agent kernel; it only handles a few basic tasks. It controls local groups and roles, manages the agent life-cycle, handles local message sending, and also offers monitoring tools. MadKit saves information about groups and roles in a replicated table in all of the hosts of the MAP.

AgentScape (Brazier et al. 2002) is a middleware system that provides support for distributed MASs. It is a research project at the Intelligent Interactive Distributed Systems Group, Section Computing Systems, Department of Computer Science, Vrije Universiteit Amsterdam.

AgentScape provides minimum support for agent applications and is configurable according to specific requirements. The current AgentScape version provides basic functionality such as the creation and deletion of agents, messaging service between agents, and weak mobility.

5 Evaluation of the multiagent platforms' performance

All of the developed experiments are explained in the following subsections. We use two PCs Intel Pentium IV @ 1.5 GHz, 256 MB of RAM memory running the Fedora Core 3 Linux operating system. Sun JDK 1.5 version is also used and MAP versions are: MadKit 4.0, AgentScape 0.8.1 and Jade 3.3.

5.1 Messaging service

In this section, we present an in-depth evaluation of messaging service for each one of the selected MAPs. First, some design details are explained in order to understand how the messaging service is implemented in each selected MAP. Then, the set of experiments performed and their results are presented. Finally, the bottlenecks detected in each messaging service implementation are shown.

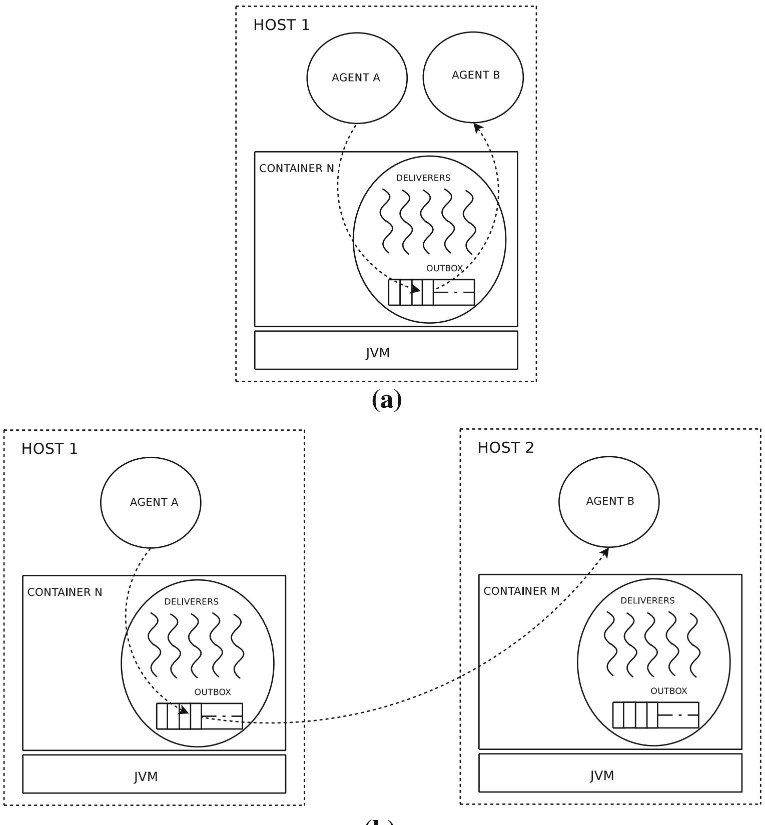


Fig. 1 Jade message sending. **a** 1 Containers. **b** 2 Containers

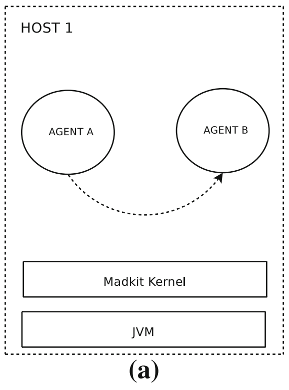
Our messaging service study does not take into account some aspects that may affect messaging performance like message ordering or the Java serialization of the messages to be sent. These aspects are future work.

5.1.1 Messaging service design details

As stated in Sect. 4, Jade is based on the container concept. All containers in a Jade MAP communicate with each other via peer-to-peer (P2P), using Java RMI. Services provided by the MAP are distributed among these containers. For each container, there is an Outbox where messages are enqueued due to a message sending request made by any one of the agents located in this container. There are also five threads in each container called, *deliverers*[5] which wait for new messages in the Outbox and perform the sending. This sending is carried out in two different ways.

If the receiver agent is located in the same container as the sender agent, the *deliverer* carrying out the sending copies the message object directly from the Outbox to the receiver agent, as can be seen in Fig. 1a.

[5] By default, there are five *deliverer* threads per container, but this can be changed when compiling the Jade MAP from Java source code to Java bytecode.



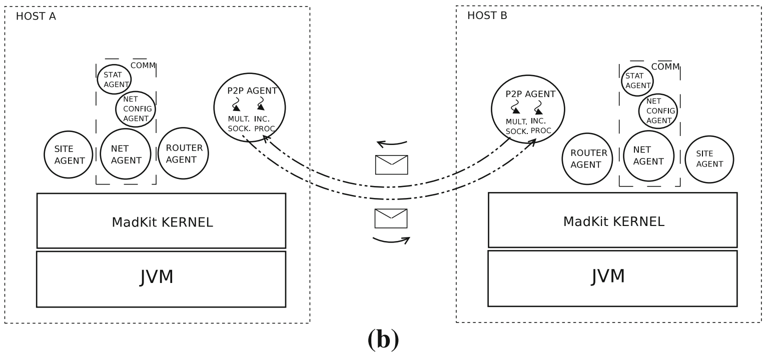


Fig. 2 Madkit message sending. **a** 1 kernel. **b** 2 kernels

If the receiver agent is located in a different container to the sender agent, the *deliverer* carrying out the sending copies the message from the Outbox to the container where the receiver agent is located via a remote procedure call using Java RMI technology. This can be seen in Fig. 1b.

In MadKit, the SiteAgent is in charge of remote kernel synchronization. The NetAgent, the NetConfigAgent, the StatAgent, the RouterAgent, the P2PAgent, and others are part of the NetComm agents. NetComm agents perform the communication of agents located in different kernels2(b). The NetAgent carries out the local kernel requests. The Router-Agent is responsible for routing messages received from the NetAgent to the P2PAgent. The P2PAgent is responsible for the connections with other kernels, and is basically composed of two threads: the Incoming Processor thread, which is in charge of the incoming messages and the MultiSock thread, which carries out the message sending to remote P2PAgents. MadKit also uses different mechanisms when sending a message depending on the location of the receiver agent of the message.

As shown in Fig. 2a, if the receiver and the sender agents are located in the same kernel, MadKit uses a direct copy-write mechanism, i.e., the thread of the sender agent copies the message object directly to the receiver agent. The NetComm agents are not involved in the message sending.

However, if the receiver and the sender agents are located in different kernels (Fig. 2b), when an agent sends a message the kernel checks where the receiver agent is placed. If the receiver agent is placed in a different kernel the steps detailed below are carried out:

1. The kernel sends the message to the SiteAgent.
2. The SiteAgent checks the kernel where the receiver agent is placed, then the SiteAgent sends the message to the local NetAgent.
3. The NetAgent sends the message to the RouterAgent which then delivers the message to a P2PAgent.
4. The P2PAgent sends the message through Java TCP sockets to a remote P2PAgent placed in the host of the receiver agent.
5. Finally, the P2PAgent delivers the message to the receiver agent.

AgentScape is composed of a *lookup service* (represented as a Python process) and a set of AgentScape Operating System (AOS) kernels distributed among several hosts. AOS Kernels in AgentScape can find each other by means of the *lookup service* that acts as a coordinating entity. Inside a single AOS Kernel, the *AgentServer* is the entity that manages the agents. By default a single *AgentServer* runs on each AOS Kernel, so every agent in the same AOS Kernel is managed by the same *AgentServer*.

When the sender of the message is running on the same AOS Kernel as the message receiver (Fig. 3a), the *AgentServer* copies the message straight to the queue of received messages of the target agent through the *Message Center* component.

When the two agents are not placed in the same AOS Kernel (Fig. 3b), there is a thread in every *AgentServer* called *MessageBuffer* which carries out the task. This thread finds the *AgentServer* where the target agent is running by means of the *lookup service* of the MAP. Thus, the message is delivered to this *AgentServer* using SunRPC. Internally, the target *AgentServer* puts the received message in the queue of received messages of the target agent. Therefore, one single thread per AOS Kernel performs the message delivery to remote agents.

The three MAPs analyzed provide different mechanisms for sending messages depending on whether the sender and receiver agents are placed in the same kernel (container in Jade terminology) or not. The three MAPs use similar mechanisms of message copies for sending messages to agents placed in the same kernel. Nevertheless, although their mechanisms are very similar, this does not mean that their performance is similar. The specific design of each messaging service is the most important feature when it comes to achieving good performance.

In order to send messages to agents placed in different kernels, the three MAPs use different technologies to contact remote objects. We can briefly see the most important features of each one:

Sockets are a service which allows programs to exchange information. TCP sockets[6] are connection oriented: once the connection is established between the two points, they are able to exchange information until the connection is closed. Java TCP Sockets are an abstraction which provides some classes that allow users to develop Java applications that require sockets TCP. Sockets can be considered a low-level communication method.

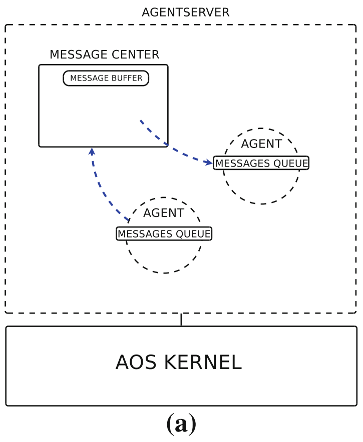
Remote Procedure Call[7] (RPC) is a high-level communication mechanism. It allows a program running on one computer to execute a function that is actually running on another computer. One of the first UNIX RPC's was developed by Sun. SunRPC[8] uses eXternal Data Representation (XDR) and messages can be sent using both TPC and UDP protocols.

Java RMI is a type of RPC mechanism. RMI can be viewed as a newer object-oriented version of SunRPC. RMI creates a stub in the client-side to communicate with the server-side

6 http://www.csc.villanova.edu/~mdamian/Sockets/TcpSockets.htm.

7 http://www.ietf.org/rfc/rfc1057.txt.

8 http://web.cs.wpi.edu/~rek/DCS/D04/SunRPC.html.



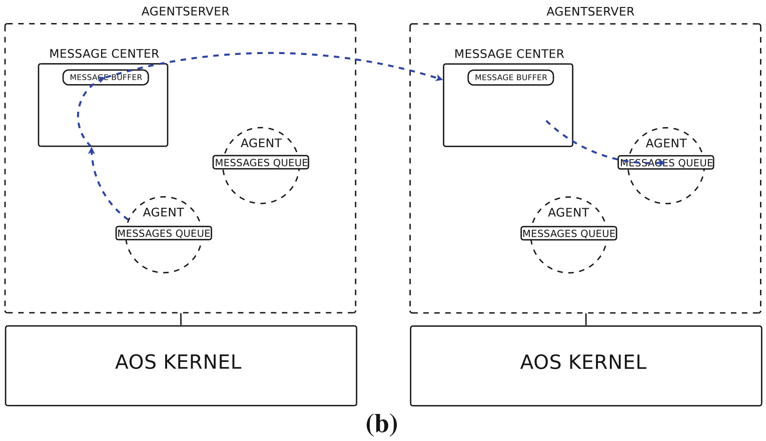


Fig. 3 AgentScape message sending. **a** 1 kernel. **b** 2 kernels

more easily. RMI is specifically designed for Java and thus, it is easier for the programmers to use.

The fastest mechanism of the three mentioned would be sockets TCP because they are a low-level mechanism. Moreover, RMI is a stronger Java object-oriented mechanism and for this reason it would perform worse than RPC or sockets. We could think that the mechanism used to develop the messaging service of any MAP is crucial to its performance, but this is not the only key issue. In our experiments the most efficient messaging service is the one provided by Jade, which uses RMI to communicate its containers. Therefore, the performance of a messaging service is not only determined by its underlying technologies but also its internal design.

5.1.2 Messaging service evaluation

As described in the introduction, a MAS is composed of agents that interact with one-another to solve a problem. All of the interactions between agents are carried out via communicating acts that involve message exchanges. Agents are not required to produce a response when they receive a message. However, measuring one way message time consumption requires host

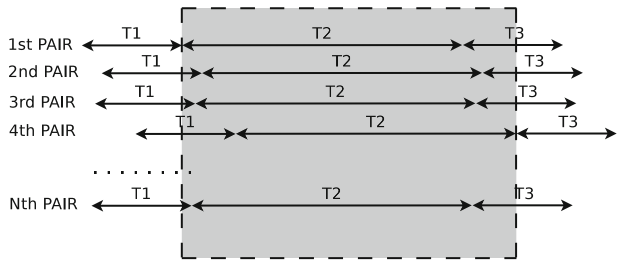


Fig. 4 Full-load measurement

synchronization when agents are placed in different hosts in such a way that times obtained are accurate enough. Therefore, we measure the Round-Trip Time (RTT) of each message sent between a sender agent and a receiver agent, i.e. the time elapsed between when the message is sent to the receiver and when it comes back to the sender so that only the time in a host is taken into account and does not require any synchronization.

Firstly, we measure the RTT between a single agent pair. Secondly, we increase the number of pairs in order to observe the MAP's scalability, and how the MAPs respond to this change.

Due to the fact that the creation and deletion of agents is not performed simultaneously, agent pairs may take their measurements in different message load conditions. As a result, the measured RTT time of each pair may vary considerably. In order to solve this problem the technique shown in Fig. 4 is used. Near each agent pair its starting and finishing time is shown. The message sending time interval for each agent pair is divided into three parts: T_1, T_2, T_3. The T_1 interval corresponds to the time that an agent pair lasts exchanging 500 messages. During this interval there is no measurement collection, the aim being to ensure that the rest of the agent pairs will be sending messages once this interval finishes. Measures are collected by each agent pair during the T_2 interval. Although there are agent pairs that start sending messages later, during their T_2 interval the rest of agent pairs will be sending messages too. Therefore, measures are taken at *full load*, i.e. there cannot be more message load in the MAP (this corresponds to the gray part of Fig. 4). Finally, once measures are collected, each pair exchanges 500 more messages (T_3) to keep the full-load conditions for all of the agent pairs that are still in their T_2 interval.

During its T_2 interval each agent pair exchanges 1,000 messages and the RTT time of each message is collected. All of the statistics are calculated from all of the RTT times for each pair taking part in the test. For instance, if the test is run with 50 pairs, statistics are calculated using 1,000 RTT times from each pair, i.e. 50,000 RTT times. Using these times we calculate the average RTT time, the RTT time 95% confidence interval, and perform Student's t tests[9] to assess whether the differences among MAPs are significant (a remark is only made when the differences are not significant).

The results obtained are processed in order to calculate an average and a standard deviation for each MAP and for each configuration. The results are presented in this paper as an average and its 95% confidence interval (calculated using the average and its standard deviation). When the results are shown graphically the 95% confidence interval can be invisible if it is too small.

[9] We perform a t test for two MAPs (Jade vs MadKit, Jade vs AgentScape and AgentScape vs MadKit) for each configuration possible in each experiment carried out with a 95% interval confidence level. We use the R software (http://www.r-project.org/) in order to perform the t tests.

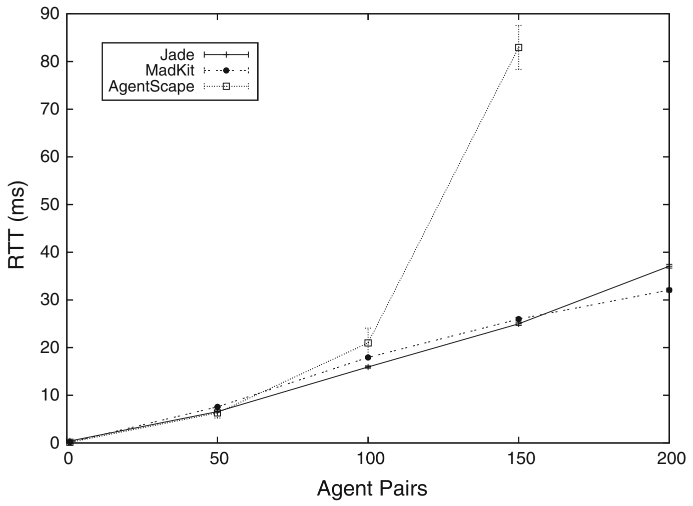


Fig. 5 Message sending, 1 host, 1 Kernel

The experiments are repeated changing the agent location. The possible locations are:

- The sender and the receiver are in the same host and live in the same kernel or container. [10]
- The sender and the receiver are in the same host but live in different kernels/containers.
- The sender and the receiver are in separate hosts (obviously in different kernels/containers).

The effect of message size on messaging service performance is also tested. For this purpose, an experiment modifying the message size (from 1 byte up to 100,000 bytes) is carried out. The number of agent pairs is fixed at 50, and the senders and the receivers are located in separate hosts.

Finally, in order to observe the MAP behavior when an agent receives many messages, we perform a test in which many agents try to communicate with a single receiver. The number of sender agents is increased from 1 to 200. The senders and the receivers are located in separate hosts.

The results plotted in Figs. 5, 6, 7, 8 and 9 show the average RTT in milliseconds and the 95% confidence interval for a given number of agent pairs communicating with each other.

Figure 5 shows the behavior of the three MAPs when the sender and receiver agents are placed in the same kernel located in a single host. Jade and MadKit have a similar response in low load (from 1 pair to 150). Moreover, when running 150 agents the differences between the two MAPs are not significant according to the *t* test performed. In contrast, when load increases (more than 150 pairs), Jade RTT grows with respect to MadKit. This may be due to the fact that MadKit uses a direct copy-write mechanism, i.e., the thread of the sender agent copies the message object directly to the receiver. In Jade, the communication in one host is based on a direct copy-write mechanism (as explained in 5.1.1) as well, but the sender agent copies its message to the Outbox, and the message remains waiting for a *deliverer*, which copies the message to the receiver agent. As a result, when load increases all of the messages in the Outbox can only be served by only five *deliverers*.

[10] The concept of container is used in Jade terminology while the concept of kernel is used in MadKit and AgentScape. However, these concepts are equivalent.

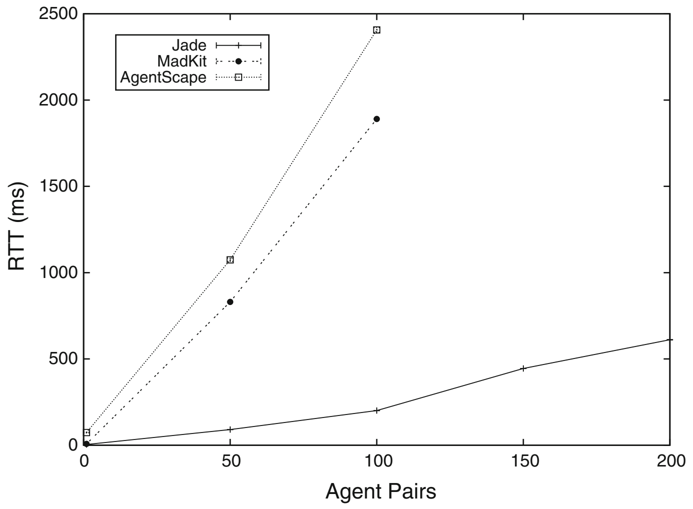


Fig. 6 Message sending, 1 host, 2 Kernels

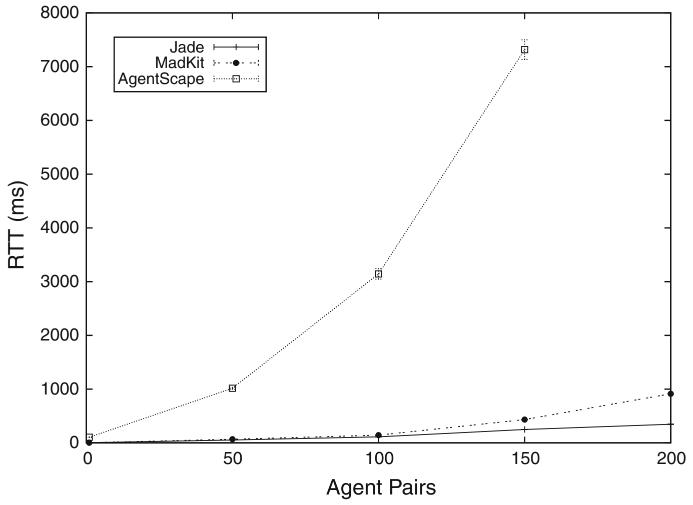


Fig. 7 Message sending, 2 hosts

AgentScape provides the poorest RTT values. When the load is low, its response is similar to or better than the other MAPs. For example, when only one pair is running the average time for sending and receiving a message is less than 0.5 ms and when running 50 pairs the differences with Jade are not significant according to the *t* test performed. However, as we increase the number of pairs, the differences in the RTT times with respect to the other MAPs also increase. This may be due to the internal kernel design of the MAP. A single *AgentServer* has to enqueue the messages to every agent managed by it. Thus, the more we increase the message traffic for a single *AgentServer*, the worse the response time is with respect to the other MAPs. It can also be observed that the confidence interval increases as the number of agents is increased, while the confidence interval for the other MAPs remains small enough

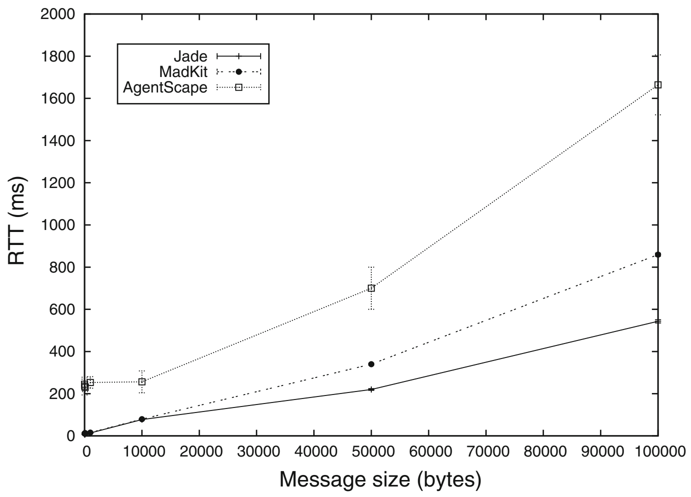


Fig. 8 Message sending 50 pairs

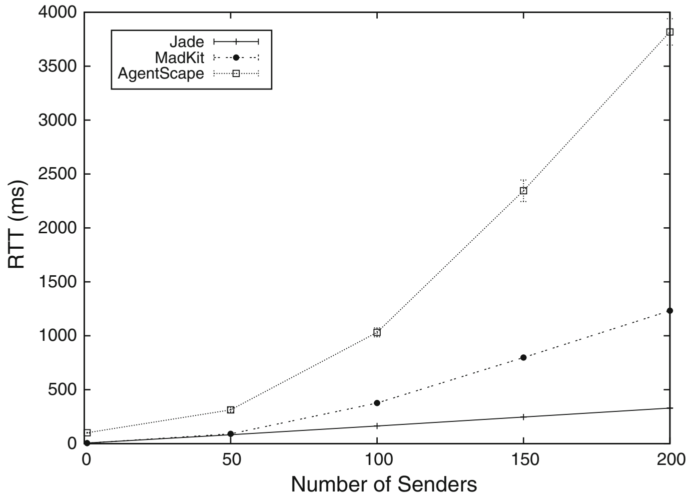


Fig. 9 Multiple sending

to be invisible in the figure. Furthermore, when we try to launch 150 pairs (300 agents in the same host), some agents get blocked and cannot send the whole amount of messages.

Observing Figs. 5, 6 and 7, it can be seen that the performance of the three MAPs is worse when the receiver and sender agents are placed in different kernels (Figs. 6 and 7). It is also easy to observe that the worst case occurs when the different kernels are not distributed between two hosts. This is because, in the distributed case, the load is shared between the two hosts and the experiments are performed in a local network.

When the agents are located in different kernels, the RTT delays in the message sending process are shorter on Jade than on the other two MAPs. As explained in Sect. 5.1.1, when the communication is made between containers, a pool of five *deliverer* threads are in charge of delivering the messages that the agents are enqueuing in the Outbox of its container. This sort of design seems to improve RTT times, with respect to the other MAPs, when more than one kernel (or container in Jade terminology) is involved, either in one host or two.

With two hosts, when running 1 agent pair, the differences between MadKit and Jade are not significant according to the *t* test performed. However, when the load increases MadKit distributed messaging causes an overload due to the set of *NetComm Agents* that are responsible for communicating different kernels through sockets. As a result, several local messages are needed to carry out distributed communication.

As stated in Sect. 5.1.1, AgentScape requires a thread (for each AgentServer) called MessageBuffer, which manages the messages when the target agent is not in the same kernel. This thread contacts the AgentServer of the target agent using SunRPC technology. In this experiment some agents get blocked if the overload of a host is greater than 150 or 200 agents.

Figure 8 shows RTT values when the message size is increased. In this experiment, 50 agent pairs were communicating with each other. AgentScape indicates the worst response times, being less adaptive to size changes than Jade and MadKit, which behaves similarly. Moreover, Jade and MadKit have no significant differences when the message size is 1,000 and 10,000 bytes according to the *t* tests performed.

Finally, the results of multiple agents sending messages to a single agent are shown in Fig. 9. In these tests Jade responses the best. It manages the message queues more efficiently than the others when many messages are received. In contrast, AgentScape and MadKit are more affected when many messages are addressed to the same agent. In this regard, the data suggest that Jade is more scalable under the tested messaging conditions.

The results obtained for the messaging service clearly show that this service is more efficient on the Jade MAP. This is mainly due to its architectural design, explained in Sect. 5.1.1. There are five dedicated threads in each Jade container in charge of sending messages, while in AgentScape and MadKit there is only one thread. As a result, having a pool of threads carrying out the message sending seems to be a good design decision. However, the level of development that has been achieved for each MAP plays a crucial role in its overall performance. AgentScape shows the poorest results, but of the three MAPs selected it is the MAP with the lowest level of development. Therefore, these bad results can be attributed not only to its architectural design but also to its level of development.

5.1.3 Messaging service bottlenecks

Once the messaging service response time has been analyzed, we want to ascertain the causes of these response times. In this section the bottlenecks detected for each messaging service implementation are presented. In order to identify the bottlenecks, experiments shown in the previous Sect. (5.1.2) are repeated, but focusing on the CPU time consumed by every component of the MAP. Thus, we can identify which threads consume more CPU time in each experiment, therefore becoming a bottleneck for that experiment.

When using a Linux operating system, we can obtain a detailed dump of every thread running in a Java Virtual Machine (JVM) through sending the signal SIGQUIT to the JVM process. In this dump, a list of Linux thread identifiers associated to each JVM thread can be obtained, including the native JVM threads and the Java threads launched by the MAP. By means of these identifiers and the *top* command we are able to calculate the CPU time of each Java thread. AgentScape also has a Python process per MAP, so, we use the Linux PID of this process to obtain CPU usage.

Due to the great number of threads involved in each experiment, the results for related threads are put together to make it easier to display them and to allow a better comparison of the MAPs tested. The *jvm* thread set refers to every thread that is in charge of the

Table 2 Message sending threads: 1 host, 1 kernel

Platform	Group	CPU time (ms)	%CPU
Jade	jvm	6334.2 +/− 36.9	14.2
	mas	25481.2 +/− 151.4	57.2
	map	391.0 +/− 4.4	0.9
	com	12372.0 +/− 154.3	27.8
AgentScape	jvm	16061.5 +/− 331.3	6.9
	mas	150625.0 +/− 3221.4	65.1
	map	38307.5 +/− 244.3	16.5
	com	26497.5 +/− 419.8	11.4
MadKit	jvm	5581.6 +/− 29.7	18.0
	mas	25375.0 +/− 163.2	81.9
	map	17.5 +/− 0.3	0.1
	com	0.0 +/− 0.0	0.0

management of the JVM. The *mas* thread set is composed of every thread of the MAS, i.e. all of the sender and receiver agents taking part in the experiment. The *com* set is composed of every thread involved in the messaging service of each MAP, i.e. the threads that take part in the communication process. Finally, the *map* set is composed of the rest of the MAP threads not associated with communication.

Experiments are repeated 100 times so that an average CPU time is calculated as well as a 95% confidence interval. Moreover, a percentage of CPU usage for each thread group is calculated from the average CPU time in order to show the consumption requirements of each thread group more clearly. Finally, Student's *t* tests are performed to assess whether the differences among MAPs are significant (a remark is only made when the differences are not significant).

The experiments sending messages between the sender and receiver agents placed in different kernels but the same host are not shown because the results obtained were practically the sum of the CPU time obtained when the sender and receiver agents are located in two hosts and did not add any more interesting information.

Table 2 shows the results obtained when 100 pairs of sender and receiver agents were running on the same host. It can be observed that the *mas* group is the most CPU demanding in the three MAPs. This is due to the fact that the *mas* group is composed of sender and receiver agents that are continuously exchanging messages with each other during the experiment. Moreover, the differences in the results of this group for Jade and MadKit are not significant according to the *t* test performed.

Both Jade and MadKit MAPs show a similar response for the *map* group, i.e. there is practically no CPU consumption by the threads of the MAP that are not related to communication purposes. However, with regards to the *com* group, MadKit and Jade behavior differs. The *com* group in MadKit has no CPU usage because, as explained in Sect. 5.1.1, when agents are located in the same kernel the thread of the sender agent copies the message object directly to the receiver agent, while in Jade the thread of the sender agent enqueues the message in the Outbox and then a *deliverer* thread performs the sending.

As one could expect, judging by AgentScape's RTT time results, its CPU times are worse than Jade and MadKit for all of the thread groups. Moreover, the *map* group in AgentScape consumes 16% of the total CPU consumption. Threads in the *map* group are supposed not to take part in the sending process, so this percentage should be lower.

Table 3 Message sending threads: 2 hosts, 2 kernels

Platform	Group	Host A		Host B	
		CPU time (ms)	%CPU	CPU time (ms)	%CPU
Jade	jvm	17636.6 +/− 190.1	9.0	16445.0 +/− 73.3	9.8
	mas	22263.8 +/− 104.5	11.4	17644.2 +/− 104.3	10.6
	map	379.6 +/− 6.2	0.2	0.0 +/− 0.0	0.0
	com	155726.2 +/− 264.5	79.4	132921.4 +/− 400.6	79.6
AgentScape	jvm	42948.5 +/− 673.1	10.6	67431.0 +/− 1895.9	10.7
	mas	14015.0 +/− 184.4	3.5	16028.5 +/− 257.1	2.5
	map	212273.0 +/− 1472.7	52.5	267236.5 +/− 3592.4	42.4
	com	135435.0 +/− 777.6	33.5	280190.0 +/− 2783.4	44.4
MadKit	jvm	31364.8 +/− 567.1	10.5	18864.3 +/− 85.9	8.5
	mas	10130.5 +/− 94.8	3.4	7812.5 +/− 57.1	3.5
	map	6944.7 +/− 22.7	2.3	7702.7 +/− 39.2	3.5
	com	250752.1 +/− 176.3	83.8	188101.9 +/− 1837.2	84.5

The results obtained when the sender and receiver agents are located in different hosts (obviously in different kernels) are shown for each MAP in Table 3. For the Jade MAP, Host A is the host on which the Main container is running, while Host B represents the host running the other container. The only major difference between them is found in the *map* group, which is composed of the *DF* and *AMS* agents, which only appear in the Main Container (these services are centralized) so that CPU time consumption of the MAP group in Host B for Jade is 0.

The *com* group increases its CPU percentage comparing to when agents are located in the same host for the three MAPs. As a result, it can be easily concluded that the threads that compound the messaging service for all of the MAPs have a greater load when more than one host is taking part in the MAP, becoming a bottleneck. However, it reaches higher values for the Jade and MadKit MAPs than for the AgentScape MAP, which requires a higher demand of CPU cycles than Jade and MadKit for all the thread groups.

5.2 Lower bound communication time

Our messaging service tests show that Jade seems to perform better than the other two MAPs tested. However, there is no way of knowing from the data presented whether the best MAP is good in absolute terms. In order to address this, we compare the measurements obtained in our tests to the time needed by simple processes when sending messages to each other. Obtaining the message sending time when simple processes are used is useful to ascertain a lower bound of the time that a messaging service could achieve.

The processes implemented communicate with each other using TCP sockets, which are the underlying technology of the technologies (Java RMI, SunRPC and Java TCP sockets) that are used by the messaging services of the three MAPs tested. There are different mechanisms for communicating processes which probably have even lower response times than shared memory or Unix sockets, but we designed this test using TCP sockets because it is the lowest level technology used in the development of the messaging services analyzed.

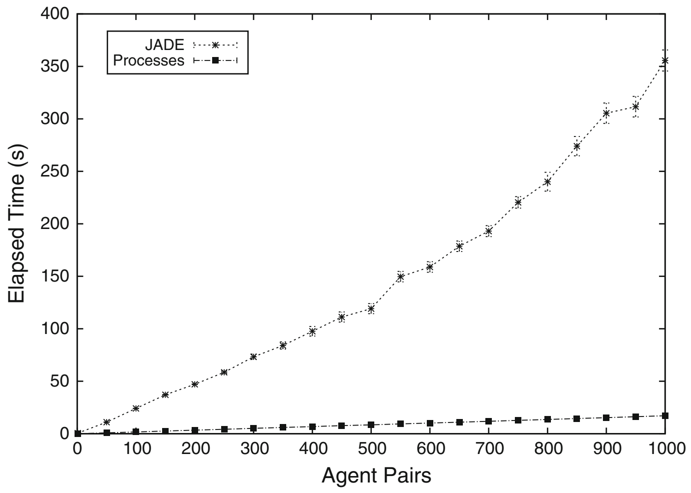


Fig. 10 Processes and Jade results in 1 host

The tests performed are similar to the ones shown in Sect. 5.1.2, but in this case, the sender and receiver agents are replaced by the sender and receiver processes. Specifically, we calculate message sending times when processes are located both in the same and in different hosts, each time increasing the number of processes taking part in the test. Moreover, the same technique is carried out to assure the same message load conditions during each test, i.e. each process pair exchanges 500 messages which are not taken into account in the measurements. After that each process pair exchanges 1,000 messages which are taken into account in the measurements, and finally, each process pair exchanges 500 more messages.

As RTT values obtained when using processes were too small and not accurate enough, for these tests we calculate the total elapsed time during the exchange of 1,000 messages when using both Jade agents or processes. Therefore, the tests are repeated 100 times so that an average, a 95% confidence interval and Student's t tests are calculated. All of the t tests performed show that the differences between the results of the two approaches tested are significant.

Figure 10 shows the results obtained when running processes on the same host and agents on the same Jade container. It can be observed that when increasing the number of agent pairs in the system, the Jade MAP performs much worse than simple processes communicating with each other using sockets, which represent the lower bound when communicating two processes using sockets.

Figure 11 shows the results obtained when placing both processes and agents in different hosts. The differences increase with respect to the results obtained when launching the test in 1 host.

When using two processes communicating each other with TCP sockets, message sending times are smaller than when using Jade agents. However, these observed times are only a lower bound when communicating two processes, because when using Jade there are no processes communicating with each other, but there are *agents* communicating with each other. Therefore, there is an unavoidable overhead caused by agent management and by the rest of the services that a MAP offers to the agents running on it. Nevertheless, such a large time difference margin makes us wonder about the fact that apart from the previously mentioned

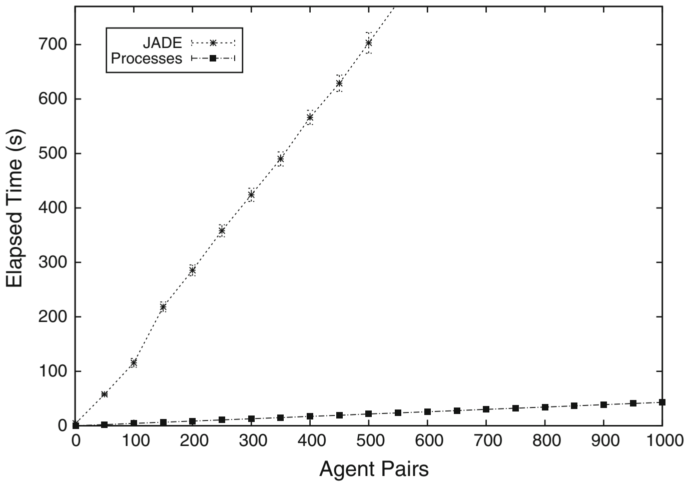


Fig. 11 Processes and Jade results in 2 hosts

unavoidable overhead, part of the overall overhead may arise when using Jade agents, caused by the layers which exist between the Operating System and an agent. This overhead would be avoidable if the design approach was different. For instance, a messaging service in which a sender agent sends a message straight to the receiver agent without centralized points so that the messaging service scales better when increasing the number of agents.

5.3 Directory service

As in Sect. 5.1, this section presents an in-depth evaluation, but in this case we focus on the directory service offered in the three selected MAPs.

5.3.1 Directory service design details

The Jade Platform offers the Directory Service by means of the DF agent. This agent is located in the Jade *Main Container*. When an agent needs to request a function from the DF agent, it sends a message containing the request following the FIPA standard, then the DF agent performs the action requested and replies to the requesting agent with the result of the action performed. Thus, at least two messages between the DF agent and the requesting agent are needed when registering, deregistering, or searching for services offered by an agent.

MadKit uses organization tables to implement the Directory Service. To register a service an agent has to request a role. MadKit kernel carries out the role registration. After role registration the kernel checks if the organization is distributed. If the organization of this role is not distributed, the role registration finishes. However, if the organization of this role is distributed among two or more kernels, the local Kernel notifies the local SiteAgent to distribute the role registration event. The local SiteAgent sends messages (using the remote sending mechanism described in 5.1.1) to every remote SiteAgent. Each remote SiteAgent updates its organization table. Role deregistration is implemented like role registration.

MadKit kernel carries out the role search. When an agent wants to know the agents that play a role, the agent asks the kernel for the information and the kernel looks up the agents that play a role in the organization table. The search action is very fast due to the fact that MadKit integrates the directory service into every kernel of the MAP.

In AgentScape, every agent can register some pairs of {name,key} to be found by other agents. This information is stored in the *lookup service* of the MAP. Every agent can register its services as pairs of {name,key} and other agents can perform searches in the *lookup service* to find these services. The *lookup service* is the centralizing entity and every register and search is done by calling the *lookup service*. Deregister is automatic, specifying the time of validity of every register in the register method call. When reaching this time since the register was made, the corresponding pair {name,key} is automatically deleted.

5.3.2 Directory service evaluation

directory service is tested by two kinds of experiments. Firstly, registration and deregistration time is measured by increasing the number of agent services already registered. The structures of the MAPs are changed from one host to two hosts in order to determine the influence of agent location. In two host experiments the agents that perform the service registrations are placed in one host, but the MAP is distributed between the two hosts. In Jade tests the Directory Facilitator is placed in the host where there is no agent and in AgentScape the *LookupService* is also placed in the host where there is no agent.

The results of service deregistration are not included because they are practically the same as the ones obtained in registration. Figures 12 and 13 depict the registration times, and Figs. 14, 15 and 16 show search times. In Figs. 12 and 13 we show the registration service time according to the number of services already registered. This time is an average time of 100 measures and a 95% confidence interval is shown as well. The 100 measures of each configuration for each MAP are used to perform Student's t tests to assess whether the differences among MAPs are significant. All of the t tests performed in the experiments in this section show that the differences among MAPs are always significant.

When registration time is taken into account, similar behavior can be observed among the three MAPs: the number of services already registered does not have any influence on a new

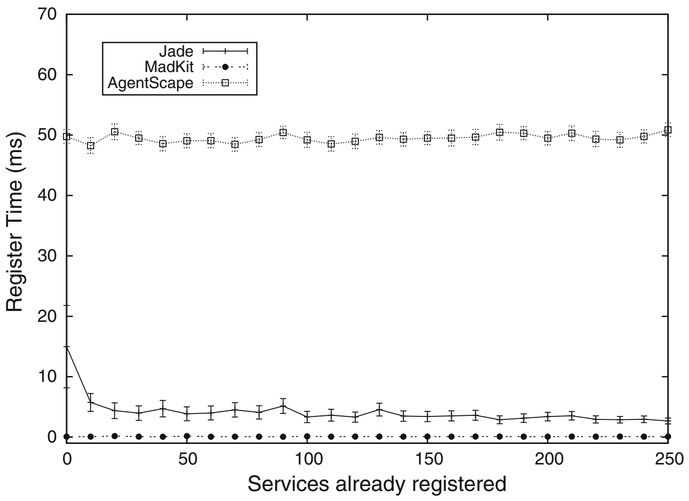


Fig. 12 Registration in 1 host

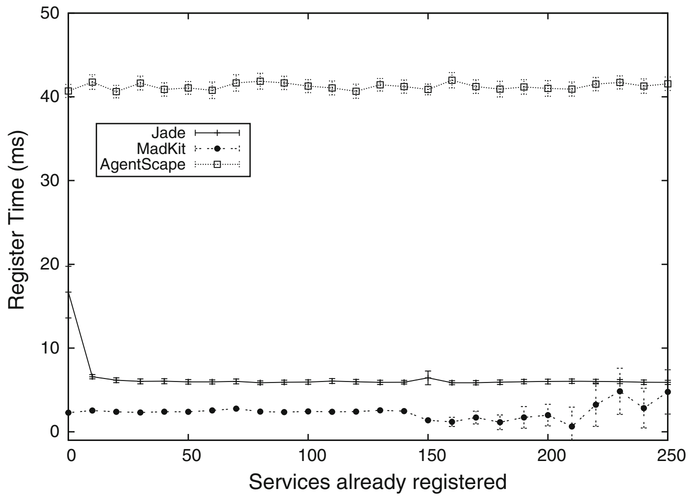


Fig. 13 Registration in 2 hosts

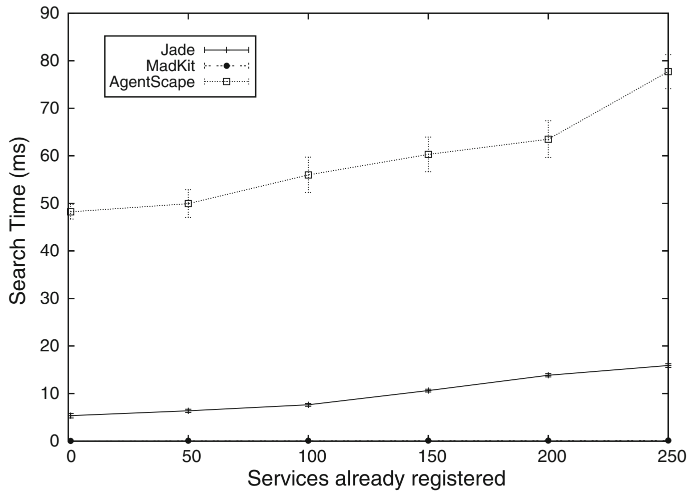


Fig. 14 Search in 1 host

service registration time. As explained in 5.3.1, the MadKit directory service is distributed and this distribution seems to improve the results obtained.

As explained in 5.3.1, in AgentScape, when an agent wants to register a new service, it must contact the *lookup service* of the MAP. If the host where this service is located has been overloaded, the process may slow down. Figures 12 and 13 show that when the agents are in a host different from the *lookup service* the process is slightly faster. This is due to the fact that *lookup service* can be placed in a host without any other components of the MAP. Hence, there is less overload in this host.

Jade's times are worse than MadKit's because agents that want to register their services must establish a communication act with the DF agent. This fact increases the sending time

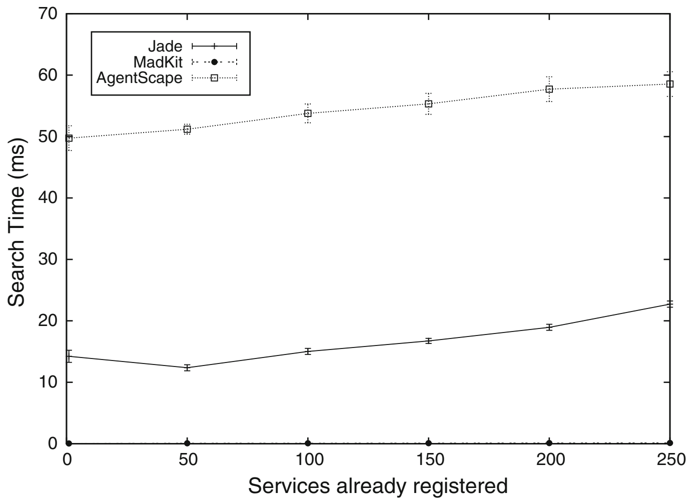


Fig. 15 Search in different hosts

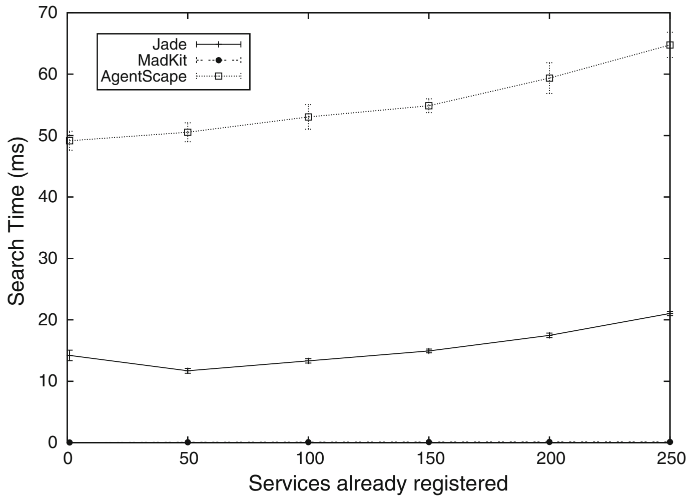


Fig. 16 Search in 2 hosts with agents distributed between them

due to ACL message exchanges. Moreover, the directory service is a centralized service in Jade (the DF agent) while in MadKit this service is distributed.

Figure 13 highlights some irregularities in MadKit registration times. These irregularities may be caused by the information replication process between the MAP hosts. Confidence interval of the three MAPs is nearly constant regardless the number of services already registered. Only MadKit increases this confidence interval when more than 200 services have been registered. We can conclude that the time for service registration in MadKit is unpredictable when replication process is required. Nevertheless, this time is always under 10 ms.

In the second experiment, we also estimate the MAP response when a service search is requested. The experiment consists of a searcher agent that performs a search for an agent offering a service. As explained in the register and deregister experiment, the number of agent services already registered is increased. The location of the agents taking part in the experiment is changed as follows:

– The searcher and registered agents are placed in the same host (Fig. 14).
– The searcher is placed in one host and the registered agents in the other host (Fig. 15).
– The searcher is placed in one host and the registered agents are distributed between the two hosts (Fig. 16).

Each experiment was repeated 100 times and the measures plotted show an average time for these times. We also show a 95% confidence interval. The significance between the results obtained for each MAP is proved by the Student's t tests performed.

In AgentScape there is a linear increase in search time. This increase in search time is caused when the *lookup service* is performing the agent's search request. It has to find the pair {name,key} requested and this time increases according the number of pairs {name,key} already registered.

Since Jade implements directory service processes using communication acts with the DF agent, its result times are not as good as those in MadKit. This is a result of how MadKit implements this service. While Jade implements it as an agent (DF agent), MadKit integrates it into the MAP software.

5.3.3 Directory service bottlenecks

In this section directory service bottlenecks are shown for each MAP using the CPU usage, as in Sect. 5.1.3. Thus, the experiments done in the evaluation of the directory service are repeated taking measurements of CPU consumption for each component of each MAP. To show the experiment results, we have grouped the threads into four groups: the *jvm* group refers the native JVM threads, which are in charge of the management of the JVM; the *mas* group is composed of every thread of the MAS (i.e. agents which register and search services); the *sds* group is composed of the specific threads of each MAP that are related with the agent search and registration services; and finally, the *map* group is composed of the rest of the internal threads of each MAP (unlike in the messaging service evaluation, the threads concerning communication are now included in this group).

As in previous experiments, Student's t tests are performed to assess whether the differences among MAPs are significant.

Table 4 shows the results for service registration in a single host for the three MAPs. The CPU time is an average time of 100 experiments with a 95% confidence interval. Student's t tests are also performed to assess whether the differences among MAPs are significant. All of the t tests performed in the experiments in this section show that the differences among MAPs are significant.

The *sds* group takes the most CPU time on the Jade MAP (almost 70%), while on the other two MAPs this percentage is lower. In Jade, this group is composed of the thread corresponding to the DF agent. This result confirms, as described in 5.3.1, that the DF agent is the bottleneck of the directory service in Jade because it centralizes the management of this service. Moreover, this agent has to serve the requests sequentially (only one thread). On the other hand, the *sds* group in MadKit hardly consumes CPU. As can be observed, the threads concerning JVM consume almost 100% of CPU time because of the fact that registering a new role in a MAP with a single host is very simple, as explained in 5.3.1. This time is practically the same as that consumed in Jade by the same group. It can be observed that AgentScape takes much more CPU time to complete the test than the other two MAPs.

Table 4 Registration threads (1 host)

Platform	Group	CPU time (ms)	%CPU
Jade	jvm	16779.6 +/– 496.1	36.1
	mas	2824.4 +/– 26.2	6.1
	map	6237.6 +/– 43.0	13.4
	sds	20638.4 +/– 67.9	44.4
AgentScape	jvm	558956.5 +/– 1511.2	47.0
	mas	48198.5 +/– 357.7	4.1
	map	224433.5 +/– 450.2	18.9
	sds	356739.0 +/– 221.1	30.0
MadKit	jvm	17231.9 +/– 116.1	96.0
	mas	532.5 +/– 4.7	3.0
	map	0.0 +/– 0.0	0.0
	sds	176.3 +/– 2.1	1.0

Table 5 Registration threads (2 hosts)

Platform	Group	Host A		Host B	
		CPU time (ms)	%CPU	CPU time (ms)	%CPU
Jade	jvm	4222.4 +/– 28.8	8.5	6374.8 +/– 55.2	21.5
	mas	0.0 +/– 0.0	0.0	3600.6 +/– 28.2	12.1
	map	23344.0 +/– 102.2	46.8	19700.6 +/– 88.1	66.4
	sds	22279.8 +/– 66.0	44.7	0.0 +/– 0.0	0.0
AgentScape	jvm	0.0 +/– 0.0	0.0	455357.4 +/– 7704.2	57.5
	mas	0.0 +/– 0.0	0.0	35332.1 +/– 2516.8	4.5
	map	0.0 +/– 0.0	0.0	218504.9 +/– 5133.3	27.5
	sds	192357.8 +/– 7581.8	100.0	83215.6 +/– 1119.7	10.5
MadKit	jvm	3614.5 +/– 12.4	7.7	4183.6 +/– 11.3	13.6
	mas	0.0 +/– 0.0	0.0	1382.1 +/– 15.4	4.5
	map	30909.2 +/– 171.2	65.5	24832.3 +/– 44.3	80.7
	sds	12693.8 +/– 83.2	26.9	367.5 +/– 6.8	1.2

We analyze another scenario when the MAP is distributed between two hosts. The results of this evaluation are shown in Table 5. In Jade we launch the main container in the host A (this container is running the DF agent) and we place the agents that are registering services in host B. The percentage of CPU used by the *sds* group is different in each container. This group in Jade is composed by the DF agent. Therefore, this time is 0 in host B. Just like in a single host, the bottleneck of this service is the DF agent, which must attend the requests related to this service. Therefore, the threads in the *map* group make higher CPU usage in two hosts than in a single one because of communication threads in charge of sending the messages needed to request registration and search actions from the DF agent.

When the MadKit MAP is composed of two hosts, the total amount of CPU time is higher than when it is composed of only one host. Moreover, it is quite similar to the total amount of CPU time consumed in Jade. In MadKit, the group that consumes more CPU time is the

Table 6 Search threads (1 host)

Platform	Group	CPU time (ms)	%CPU
Jade	jvm	1619.6 +/− 7.9	39.7
	mas	217.2 +/− 8.1	5.3
	map	589.8 +/− 10.2	14.4
	sds	1656.0 +/− 8.8	40.6
AgentScape	jvm	4350.8 +/− 32.1	17.8
	mas	786.0 +/− 9.3	3.2
	map	11303.8 +/− 42.1	46.3
	sds	7928.6 +/− 25.2	32.6
MadKit	jvm	1184.3 +/− 13.2	29.6
	mas	2798.6 +/− 17.3	69.9
	map	0.0 +/− 0.0	0.0
	sds	18.2 +/− 0.4	0.5

map group. As described in 5.3.1, registering a new role in a distributed MAP needs the distribution of its organization information. For this reason, communication threads (which are included in *map* group) consume the most CPU time. Again the messaging service is the bottleneck of the MAP.

In AgentScape we launch the *lookup service* in host A and the agents in host B. Because of AgentScape design 5.3.1, we can launch the *lookup service* in a host without requiring a kernel. Thus, in host A there is only CPU usage by *sds* group. This usage concerns the CPU time of the *lookup service* python process. In host B we launch the AOS kernel the agents are running on.

Below, we are going to detail the results of the experiments for the service search. In these tests we have launched 200 agents which offer a service in the same host or divided into two hosts, and a searcher agent which searches a service and takes the CPU time measurements.

Table 6 shows the results of search experiments on the three MAPs when agents that offer services are placed in the same host as the agent which searches for a service. As in register experiments, we show an average time of 100 repetitions with a 95% confidence interval. AgentScape is the MAP which consumes the most CPU time. In Jade, the *sds* group is a bottleneck again because the DF agent is included in this group and searching for service is similar to registering a service: agents have to contact the DF agent. Nevertheless, searching is faster than registering and for this reason, the *jvm* group time is quite similar to the *sds* CPU time. In AgentScape, the *map* group consumes more CPU time than both the *mas* and *sds* groups. As in Jade, searching is faster than registering. So in this case, the *jvm* CPU time is considerably lower than when registering, but the *lookup service* does not become a bottleneck with this amount of load. In MadKit, the most CPU time is consumed by *mas* group. This CPU usage is higher than the one obtained in the register tests. This is due to the great amount of requests made by the searcher agent.

In the case of two hosts we launch 100 agents, each one registering a service, placing the searcher agent in host B. Results are presented in Table 7. As in the register experiments, in Jade MAP, host B does not consume CPU time in *sds* group because the DF service is placed in host A. In MadKit, when launching the agents in two kernels we can observe that the host where the searcher agent is placed behaves in a similar way to the case of one host, due to the fact that the searcher agent is the agent which uses the most CPU time. In this case, the *map* group has CPU usage because distribution of the organization information is required.

Table 7 Search threads (2 hosts)

Platform	Group	Host A		Host B	
		CPU time (ms)	%CPU	CPU time (ms)	%CPU
Jade	jvm	1696.0 +/– 10.6	39.6	969.4 +/– 7.4	50.3
	mas	10.0 +/– 1.6	0.2	189.6 +/– 6.7	9.8
	map	1216.2 +/– 14.5	28.4	768.4 +/– 9.6	39.9
	sds	1361.4 +/– 12.4	31.8	0.0 +/– 0.0	0.0
AgentScape	jvm	2738.2 +/– 26.0	20.5	2868.8 +/– 22.7	27.9
	mas	0.0 +/– 0.0	0.0	220.2 +/– 4.1	2.1
	map	5512.2 +/– 39.5	41.2	5166.0 +/– 31.5	50.2
	sds	5128.6 +/– 19.2	38.2	2041.6 +/– 37.9	19.8
MadKit	jvm	2419.7 +/– 21.2	70.4	2989.7 +/– 23.4	40.1
	mas	0.0 +/– 0.0	0.0	4031.2 +/– 31.5	54.1
	map	628.3 +/– 5.9	18.3	356.1 +/– 3.8	4.8
	sds	386.9 +/– 4.2	11.3	72.8 +/– 1.2	1.0

For this reason, communication threads are needed. As in one host, we can notice that in AgentScape, the *map* is the most influential group. Although the overload caused by the *jvm* is significant, it does not consume the most CPU time. Moreover, the *mas* group does not consume much more CPU time, which is practically consumed by the searcher agent placed in host B. The CPU usage of the *sds* group is much higher in host A due to the fact that the *lookup service* is running.

5.4 Consumed memory

We have also evaluated how much memory every Java thread uses because the agents of each MAP are implemented as a Java thread. For this reason, we calculate the memory consumed by the JVM without the MAP, the memory taken up by each MAP, and finally, the memory used by an idle agent inside each MAP. We use both the Java API and the application *jconsole* which shows information about memory and other parameters of the JVM processes by means of an interface.

To perform the tests we use two methods of the `java.lang.Runtime` class:

– `totalMemory()`, returns the total memory available in the JVM.
– `freeMemory()`, returns the free memory available in the JVM.

To estimate JVM usage we create a Java object that only executes a few code lines and uses the two instructions explained previously.

After repeating 100 times we can observe that the JVM (and actually this light object) uses 176 KB of memory without any variance in the 100 repetitions.

The next step is to find out the memory usage of each MAP. This can be seen it in Table 8. We execute each MAP over JVM and then calculate its memory usage in a similar way to the previous test, i.e. launching an agent to measure these parameters and repeating it 100 times, but in this case there is variance among repetitions so that an average and a 95% confidence interval is calculated. Jade is the lighter MAP, only using 940 KB, while AgentScape consumes more than 5,000 KB.

Table 8 Memory usage

Platform	Platform mem. usage (KB)	100 Agent mem. usage (KB)
Jade	940.52 +/− 3.46	1102.85 +/− 6.78
MadKit	1294.21 +/− 8.34	1923.38 +/− 11.42
AgentScape	5218.35 +/− 57.46	26979.37 +/− 342.52

Table 8 also shows memory usage of 100 agents. For each MAP, we implement a MAS formed by 100 idle agents. We launch every agent and then take the memory usage repeating it 100 times. This amount includes the memory of the JVM, the MAP and the 100 agent system together. To obtain the memory usage of one agent we could subtract the JVM and the MAP memory usage. In these tests Jade also achieves the bests result.

The results obtained for memory usage prove that memory usage is not a bottleneck in current MASs because in the worst case of the three MAPs (AgentScape) the jvm plus the MAP plus 100 agents use 27MB.

5.5 Network occupancy rate

Another important issue to be evaluated is whether network connections are involved in the decrease in performance when increasing the load of the MAPs. For this purpose we designed a set of experiments with many message traffic to measure the network occupancy rate.

In these experiments we launched two kernels in two hosts, running receiver agents in the first one and sender agents in the second one, gradually increasing the number of agents launched. Each pair sends 1,000 messages itself, collecting the parameters referring to the network. These experiments allow us to show if the network capacity is an influential issue in the scalability of the MAPs. We used the *iptraf* program to do this test.

Iptraf is a piece of software included in some Linux distributions (Fedora, Debian, SuSE, Conectiva, RedHat, ...). It shows network statistics like byte counters, TCP connection packages, network interface statistics, TCP/UDP traffic breakdowns and LAN station bytes and packages. It provides us with the network throughput for each MAP while increasing the number of agent pairs communicating. These network throughputs are collected and plotted.

Figure 17 shows the test results of the three MAPs. In Jade tests, the network throughput increases when running up to 150 pairs. From this number of agent pairs on, the network throughput remains constant at about 8,200 Kbits/s. We can assume that 8,200 Kbits/s is the upper limit of the Jade Message Transport System which is composed of 5 *deliver* threads in each Container of the MAP. MadKit's upper limit is 4,900 Kbits/s. When communicating agents are more than 50 pairs the MadKit network throughput goes down to 900 Kbits/s. This blockage is due to the communicator agents. These agents make a new socket connection for each message sending, when the number of agent pairs increases, the threads running in the same computer are higher and communicator thread delivering time increases. AgentScape test's network throughput is always below 1,000 Kbits/s in this version. In the tests where there are few agent pairs the troughput remains low, when the agent pairs are increased the network throughput remains around 1,000 Kbits/s. AgentScape message delivery structure does not allow for higher network traffic.

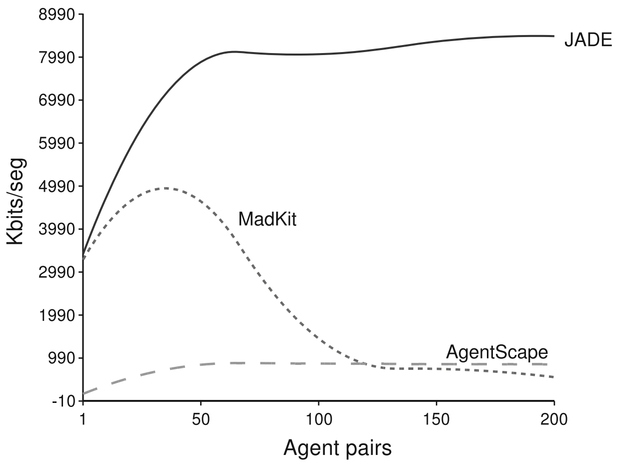


Fig. 17 Network throughput

Consequently, network capacity is not a bottleneck because every MAP test throughput is much lower than network maximum throughput, which is 100,000 Kbits/s (Fast Ethernet).

6 Conclusions

This article is aimed at determining the extent to which the internal design of a MAP affects its performance. As can be observed in the results obtained, three different internal designs of a MAP lead to three different performances when running the experiments presented. These experiments mainly evaluate the response time of these three MAPs when changing parameters like message traffic, the amount of agents running, and so on. These experiments draw a common conclusion: all the three MAPs perform poorly and demonstrate low scalability when the MAS being run on increases. Therefore, performance is a key issue to be taken into account when designing a MAP.

We can also conclude from the results of the experiments performed:

- The Jade messaging service performs better than the messaging service of the other two MAPs. Therefore, from the three MAPs tested, MAS developers should use Jade if they are concerned about efficiency. The key issue in the Jade messaging service design is the use of a pool of threads (the *deliverers*) in charge of sending the messages enqueued in each container, while in MadKit, a message is sent through a chain of communication agents (NetAgent, RouterAgent, P2PAgent, and so on). It seems that when increasing the overall message traffic in the MAP, this agent chain becomes a clear bottleneck. AgentScape messaging service design is similar to the Jade one, but it only uses a single MessageBuffer thread for sending messages to remote kernels, becoming a bottleneck when increasing the message traffic. Nevertheless, AgentScape poor performance could also be attributed to its early development stage.
- Although the Jade messaging service performs better than the other two messaging services, it may become a bottleneck if the message traffic is very high. This conclusion is based on the differences observed with respect to simple processes communicating to each other via TCP sockets in Sect. 5.2. Therefore, in order to design a messaging

service for a MAP that can handle large agent populations, the messaging service design should be based on direct communication between each pair of agents. Messages are used for communication between agents and in many MAPs, also for interacting with other services. Therefore, the design of this service is crucial in the performance of the MAP.

– The directory service is not a bottleneck in the MAP performance. Registering a service lasts a constant time in the three analyzed MAPs, regardless of the number of agents running on the MAP and the services already registered. Searching for a service can increase its response time according to the amount of services registered (only in Jade and in AgentScape), but this increment is so small that we are inclined to believe that it should not become a bottleneck.The best directory service design seems to be the one implemented in MadKit, in which this service is provided by the MAP, in a distributed way. In Jade and in AgentScape this service is centralized. What is more, in Jade this service is provided by the DF agent so that communication acts are needed. Therefore, we can conclude that services like directory service, which stores information and receives a lot of requests, should be distributed among the various hosts in the MAP whenever possible, for improving the response time of these services.

References

Ahmad HF, Suguri H, Ali A, Malik S, Mugal M, Shafiq MO, Tariq A, Basharat A (2005) Scalable fault tolerant agent grooming environment: Sage. In: AAMAS ’05: proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, ACM Press, New York, pp 125–126, http://doi.acm.org/10.1145/1082473.1082816

AOS Group (2008) An agent infrastructure for providing the decision-making capability required for autonomous systems. http://www.agent-software.com

Argente E, Julian V, Botti V (2005) From human to agent organizations. In: CoOrg-05: proceedings of the first international workshop on coordination and organisation, pp 1–11

Bellavista P, Corradi A, Stefanelli C (1999) A secure and open mobile agent programming environment. Autonomous decentralized systems, international symposium on 0:238, http://doi.ieeecomputersociety.org/10.1109/ISADS.1999.838439

Bellifemine F, Caire G, Poggi A, Rimassa G (2003) Jade a white paper. Telecom Italia EXP Mag 3(3):6–19

Bellifemine F, Caire G, Poggi A, Rimassa G (2008) Jade: a software framework for developing multi-agent applications. Lessons learned. Inform Softw Technol 50(1–2):10–21

Bigus JP, Schlosnagle DA, Pilgrimand JR, Mills WN III, Diao Y (2002) Able: a toolkit for building multiagent autonomic systems. IBM Syst 41:350–371

Bitting E, Carter J, Ghorbani A (2003) Multiagent system development kit: an evaluation. In: Proceedings of communication networks and services research conference, May 15–16, Moncton, pp 80–92

Bordini RH, Wooldridge M, Hübner JF (2007) Programming multi-agent systems in AgentSpeak using Jason (Wiley Series in Agent Technology). Wiley, New York

Brazier F, Mobach D, Overeinder B, van Splunter S, van Steen M, Wijngaards N (2002) Agentscape: middleware, resource management, and services. In: Proceedings of the 3rd international SANE conference, pp 403–404

Bäumer C, Breugst M, Choy S, Magedanz T (1999) Grasshopper—a universal agent platform based on OMG MASIF and FIPA standards. In: Karmouch A, Impey R (eds) Mobile agents for telecommunication applications, proceedings of the first international workshop (MATA 1999), October 1999, World Scientific Pub, Ottawa, pp 1–18

Burbeck K, Garpe D, Nadjm-Tehrani S (2004) Scale-up and performance studies of three agent platforms. In: IPCCC 2004

Camacho D, Aler R, Castro C, Molina JM (2002) Performance evaluation of zeus, jade, and skeletonagent frameworks. In: Systems, man and cybernetics, 2002 IEEE international conference on

Chmiel K, Tomiak D, Gawinecki M, Karczmarek P (2004) Testing the efficency of jade agent platform. In: Proceedings of the ISPDC/HeteroPar’04, 49–56

Collis JC, Ndumu DT, Nwana HS, Lee LC (1998) The zeus agent building tool-kit. BT Technol J 16(3):60–68

Contreras M, Germán E, Chi M, Sheremetov L (2004) Design and implementation of a fipa compliant agent platform in.net. J Object Technol 3(9):5–28

Cortese E, Quarta F, Vitaglione G (2003) Scalability and performance of jade message transport system. EXP 3:52–65

Cost RS, Finin T, Labrou Y, Luan X, Peng Y, Soboroff I, Mayfield J, Boughanam A (1998) Jackal: A Java-based Tool for Agent Development. In: AAAI-98, workshop on tools for agent development, Madison, WI

Dale J (2002) April agent platform reference manual. Fujitsu Laboratories of America

Duvigneau M, Moldt D, Rölke H (2003) Concurrent Architecture for a multi-agent Platform. In: Giunchiglia F, Odell J, Weiß G (eds) Agent-oriented software engineering III, vol 2585. Third international workshop, agent-oriented software engineering (AOSE) 2002, Bologna, Italy, July 2002. Revised Papers and Invited Contributions, Springer, Berlin, LNCS

Escriva M, Palanca J, Aranda G, Garca-Fornes A, Julian V, Botti V (2006) A jabber-based multi-agent system platform. In: Proceedings of the fifth international joint conference on autonomous agents and multi-agent systems (AAMAS06), Association for Computing Machinery, Inc., ACM Press, New York, pp 1282–1284

FIPA (2000) The foundation for intelligent physical agents. http://www.fipa.org

Giang NT, Tung DT (2002) Agent platform evaluation and comparison

Graham JR, Decker K, Mersic M (2003) Decaf—a flexible multi agent system architecture. Auton Agents Multi-Agent Syst 7(1–2):7–27

Gray RS (1995) Agent Tcl: a transportable agent system. In: Proceedings of the CIKM workshop on intelligent information agents, fourth international conference on information and knowledge management (CIKM 95), Baltimore

Gutknecht O, Ferber J (2000) The madkit agent platform architecture. Lect Notes Comput Sci 1887:48–55

Helsinger A, Thome M, Wright T (2004) Cougaar: a scalable, distributed multi-agent architecture. In: SMC(2), IEEE, pp 1910–1917

Inc IA (2004) User guide. http://www.opencybele.org/docs/UsersGuideCybeleProVersion1.0.pdf

Ita ME (1997) Concordia: an infrastructure for collaborating mobile agents

Jarvinen J (2002) Agentdock platform bdi-agents. http://www.cs.uta.fi/kurssit/AgO/ago7a-print.pdf

Kusek M, Voncina D, Vyroubal V (2004) Design and implementation of the mobile agent platform crossbow. In: Proceedings of the conference CTI—telecommunications & information, pp 82–87

Lee LC, Ndumu DT, Wilde PD (1998) The stability, scalability and performance of multi-agent systems. BT Technol J 16:94–103

Luck M, McBurney P, Shehory O, Willmott S (2005) Agent technology: computing as interaction (A Roadmap for Agent Based Computing). AgentLink

Lugmayr W (1999) Gypsy: a component-oriented mobile agent system. URL: citeseer.ist.psu.edu/lugmayr99gypsy.html

Meyer AP (2004) A multi-agent systems engineering environment for the semantic web

Minar N, Gray M, Roup O, Krikorian R, Maes P (1999) Hive: distributed agents for networking things. In: Proceedings of ASA/MA'99, the first international symposium on agent systems and applications and third international symposium on mobile agents

Mulet L, Such JM, Alberola JM (2006) Performance evaluation of open-source multiagent platforms. In: Proceedings of the fifth international joint conference on autonomous agents and multiagent systems (AAMAS06), Association for Computing Machinery, Inc., ACM Press, New York, pp 1107–1109

Nwana H (1994) Negotiation strategies: an overview. Internal report 14, BT laboratories

Omicini A, Rimassa G (2004) Towards seamless agent middleware. In: TAPOC 2004

Peine H, Stolpmann T (1997) The architecture of the ara platform for mobile agents. In: First international workshop on mobile agents (MA 97)

Ricordel PM, Demazeau Y (2000) From analysis to deployment: a multi-agent platform survey. In: ESAW'00, engineering societies in the agents' world

Riekki J, Huhtinen J, Ala-Siuru P, Alahuhta P, Kaartinen J, Roning J (2003) Genie of the net, an agent platform for managing services on behalf of the user. Comput Commun 26(11):1188–1198 (ubiquitous Computing)

Roth V, Jalali-Sohi M (2001) Concepts and architecture of a security-centric mobile agent server. In: ISADS

Shakshuki E (2005) A methodology for evaluating agent toolkits. In: ITCC '05: proceedings of the international conference on information technology: coding and computing (ITCC'05)—volume I, IEEE Computer Society, Washington, DC, pp 391–396, http://dx.doi.org/10.1109/ITCC.2005.15

Shi Z, Zhang H, Cheng Y, Jiang Y, Sheng Q, Zhao Z (2004) Mage: an agent-oriented programming environment. In: ICCI '04: proceedings of the third IEEE international conference on cognitive informatics, IEEE Computer Society, Washington, DC, pp 250–257, http://dx.doi.org/10.1109/ICCI.2004.20

Silva L, Soares G, Martins P, Batista V, Santos L (2000) Comparing the performance of mobile agent systems: a study of benchmarking. Comput Commun 23:769–778

Software R (2007) Voyager messaging developer's guide. http://www.recursionsw.com/
Ten Hoeve EC (2003) 3APL Platform. Master's thesis, Utrech University
Tripathi AR, Karnik NM, Ahmed T, Singh RD, Prakash A, Kakani V, Vora MK, Pathak M (2002) Design of the ajanta system for mobile agent programming. J Syst Softw 62:123–140
Venners B (1997) The architecture of aglets. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html
Vrba P (2003) Java-based agent platform evaluation. In: Proceedings of the HoloMAS 2003, pp 47–58
Wooldridge M (2002) An introduction to multiagent systems. Wiley, England
Wooldridge M, Jennings NR (1995) Intelligent agents: theory and practice. Knowl Eng Rev 10(2):115–152
Xu H, Shatz SM (2003) Adk: an agent development kit based on a formal design model for multi-agent systems. J Autom Softw Eng 10:337–365