Achieving Mobile Agent Systems interoperability through software layering ☆

Giancarlo Fortino *, Alfredo Garro, Wilma Russo

Dipartimento di Elettronica, Informatica e Sistemistica (DEIS), Università della Calabria, Via P. Bucci, 87036 Rende (CS), Italy

Received 20 April 2006; received in revised form 5 February 2007; accepted 15 February 2007
Available online 25 February 2007

Abstract

Interoperability is a key issue for a wider adoption of mobile agent systems (MASs) in heterogeneous and open distributed environments where agents, in order to fulfill their tasks, must interact with non-homogeneous agents and traverse different agent platforms to access remote resources. To date, while several approaches have been proposed to deal with different aspects of MAS interoperability, they all lack the necessary flexibility to provide an adequate degree of interoperability among the currently available MASs. In this paper, we propose an application-level approach grounded in the software layering concept, which enables execution, migration and communication interoperability between Java-based mobile agent systems, thus overcoming major setbacks affecting the other approaches currently proposed for supporting MAS interoperability. In particular, we define a Java-based framework, named JIMAF, which relies on an event-driven, proxy-based mobile agent model and supports interoperable mobile agents which can be easily coded and adapted to existing MASs without any modifications of the MAS infrastructures. Results from the performance evaluation of MAS interoperability was carried by using JIMAF atop Aglets, Ajanta, Grasshopper, and Voyager, demonstrating that the high-level JIMAF approach offers high efficacy while maintaining overhead at acceptable levels for target computing environments.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Mobile Agent Systems; Interoperability; Agent Models; Proxy-based Communication; Frameworks; Software Layering

1. Introduction

Over the last decade, mobile agent paradigm and technology [14,18,6] have shown high potential and flexibility for the design and implementation of general-purpose distributed applications, particularly in dynamic environments such as those based on the Internet. A mobile agent can be defined as an autonomous software entity which can migrate from one host to another to fulfill a goal-oriented task. Mobile agents are supported and managed by distributed software platforms known as mobile agent systems (MASs). Although plenty of MASs are currently available and several critical advantages deriving from the exploitation of mobile agents have clearly been demonstrated by both academic and industrial research efforts [21,14,6], the following major drawbacks still hinder the widespread use of mobile agents: the lack of appropriate security mechanisms, the lack of interoperability, and the lack of software development processes and modeling languages tailored for mobile agents.

In this paper, we focus on interoperability which is extremely important and necessary when heterogeneous MASs come into contact with each other. Such contacts can be very frequent in open systems or multi-domain environments whereby a mobile agent, in order to fulfill its goal, must cross several domains served by MASs of different kinds to access remote resources and/or communicate with non-homogeneous mobile/stationary agents.

* A preliminary version of this article, entitled “High-level Interoperability between Java-based Mobile Agent Systems” by G. Fortino and W. Russo, was presented at the 17th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS’04), San Francisco, CA, USA, pp. 367–374, 15–17 September 2004.

* Corresponding author. Tel.: +39 0984 494063; fax: +39 0984 494713.
E-mail addresses: g.fortino@unical.it (G. Fortino), garro@unical.it (A. Garro), w.russo@unical.it (W. Russo).

0950-5849/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.02.016
Interoperability between MASs can be defined as follows: two MASs are interoperable if a mobile agent of one system can migrate into the second heterogeneous system, execute on this system, and communicate with other co-located or remote mobile agents [29]. According to this definition, interoperability among MASs involves the following issues:

- **Execution**: How to adapt the lifecycle of a mobile agent to different MASs and how to enable a mobile agent to access resources of MASs.

- **Inter-agent communication**: How to adapt the message transport layers of different MASs to one another and the communication language among different mobile agents.

- **Mobility**: How to adapt different agent transport and agent encoding protocols among different MASs.

- **Security**: How to authenticate mobile agents of different MASs and appraise their behaviour.

To date several approaches have been proposed for tackling the interoperability issues which can be classified as **specification-based**, **low-level**, or **high-level** approaches. However, such approaches are still not satisfactory mainly due to the following limitations: (i) only some of the aforementioned interoperability issues are addressed; (ii) low effectiveness for delivering low-cost and flexible solutions. Moreover, such approaches lack documentation about performance overhead regarding how the MASs have been made interoperable.

In this paper, we propose a **high-level** approach grounded on the well-established concept of software layering [20], which is able to provide interoperability of execution, communication and migration, and overcome the limitations of the currently available solutions. Moreover, security remains under the control of specific mechanisms of the MASs which are made interoperable. As our approach also relies on voluntary interoperability, MASs which wish to interoperate must adopt a common solution as the use of a unifying framework or the exploitation of common supporting tools [29].

Software layering has been largely used for the development of communication protocol suites to hide network heterogeneity: TCP/IP is the most notable example. Therefore to hide heterogeneity of Java-based MASs we have designed and implemented a software layer, the Java-based Interoperable Mobile Agent Framework (JIMAF), which can be layered atop available Java-based MASs (see Fig. 1) without requiring any modifications to the underlying MASs.

In particular, the framework is structured in three sub-layers: (i) the Interoperable Mobile Agent Layer (IMAL), which supports the programming of mobile agents which are independent from any specific MAS; (ii) the Adaptation Layer (AL), which adapts an interoperable mobile agent to a specific MAS; (iii) the Platform-dependent Mobile Agent Layer (PMAL), which provides a set of basic, platform-dependent mobile agents on which interoperable mobile agents can be adapted via the AL. An interoperable mobile agent therefore consists of a MAS-neutral **high-level** part which does not change throughout the mobile agent lifecycle, and a **low-level** part which depends on the specific MAS in which the mobile agent is operating and therefore changes after a heterogeneous migration, assuming the “shape” of the mobile agent of the new hosting MAS. The **high-level** part of an interoperable mobile agent is programmed according to a programming model based on the concept of event-driven lightweight agent [9,10].

An application based on JIMAF is composed of (i) an interoperable part written according to JIMAF (i.e. interoperable mobile agents able to migrate, execute, and communicate atop each platform actually and potentially made interoperable by JIMAF) and, possibly, (ii) a platform-dependent part written according to the specific platforms (i.e. Aglets agents, Voyager agents, etc.) and integrated with the interoperable part through wrapper agents which are special JIMAF agents wrapping platform-dependent agents which are thus capable of communicating with both interoperable agents and with specific platform-dependent agents. Thus, communication between an interoperable mobile agent and a platform-dependent mobile agent is mediated by wrapper agents, whereas communication among interoperable mobile agents is based on asynchronous messages. Moreover, migration of interoperable mobile agents between two heterogeneous MASs is enabled by special JIMAF agents called bridges which are software equivalents of network bridges.

Currently, the framework is layered atop the following MASs: Aglets, Ajanta, Voyager and Voyager. These MASs were selected from the large number of available Java-based MASs as they are widely cited and very well documented, robust and stable, very representative of ongoing research and commercial efforts, extensively used for developing a broad range of applications as well as sophisticated programming systems.

In order to evaluate the performance of JIMAF over the MASs currently exploited, so to quantify performance overhead which an application-level approach inevitably introduces, several micro-benchmarks and a micro-kernel were defined and run to evaluate the performances of...
migration, lookup, message transmission, creation, and roaming of the interoperable mobile agent and to compare the results with those of the mobile agents of the MASs atop which JIMAF was layered. A contained performance overhead was found which is acceptable within Internet-based heterogeneous computing environments where interoperability is often more important than high performances [9,14].

The main contributions of this paper are the following:

- The design and implementation of the model-driven high-level framework, JIMAF, which renders Java-based mobile agent systems interoperable without requiring their modification.
- The evaluation of the performance of JIMAF with respect to those of the MASs made interoperable (Ajanta, Aglets, Grasshopper, and Voyager).
- The introduction and exploitation of a taxonomy to present the state-of-the-art regarding MAS interoperability and discuss the JIMAF proposal.

The remainder of the paper is structured as follows. Section 2 introduces background concepts regarding MASs through a reference framework. Section 3 describes the fundamentae of our approach: the interoperable mobile agent model, the communication infrastructure and the bridging architecture. Section 4 details the Java-based interoperable mobile agent framework along with an example which shows the main JIMAF features. The performance evaluation of JIMAF is reported in Section 5. Section 6 presents related work in the area of MAS interoperability with reference to the introduced taxonomy and discuss our proposal in the context of the high-level approaches. Finally, conclusions are drawn and directions of future work briefly delineated.

2. Background

A Mobile Agent System (MAS) is a distributed software system responsible for supporting and managing mobile agents, i.e. autonomous software entities which can migrate from one host to another to fulfill a goal-oriented task. The research interest in MAS technology has largely been driven by the exploitation of Java as a commonly available mobile code technology and platform [18]. In addition to code mobility, Java provides a security architecture that is crucial in constructing solutions for security problems in MASs. A multitude of Java-based MASs have been built over the last few years and although all of them share the same Java-enabled basic characteristics such as weak mobility, network class loading on demand, communication mechanisms based on message passing, they are significantly different and not interoperable.

In order to identify both the differences and the commonalities among Java-based MASs, several MASs were analyzed; these MASs were selected according to the following criteria: (i) widely cited; (ii) robust and stable; (iii) very representative of on-going research and commercial efforts; (iv) very well documented; (v) extensively used for developing a broad range of applications and sophisticated programming systems. This analysis provided architectural, programming, design patterns on which the JIMAF framework proposed in this paper was engineered. In particular, the following MASs were considered:

1. Aglets Workbench 2.0.2 or simply “Aglets” [1,21] is a MAS developed by the IBM Tokyo Research Laboratory, and is probably the most well-known Mobile Agent System.
2. Ajanta 2.0 [2,33] is a MAS developed at the University of Minnesota whose main purpose is to serve as an infrastructure for research in secure distributed programming using mobile agents.
3. Grasshopper 2.2.4 [12], which was jointly developed by GMD Focus and IVK++ GmbH, is the first MAS which is compliant with the OMG’s MASIF [24,25] and FIPA [11] specifications.
4. Voyager ORB 3.2 [27] is a full-fledged ORB system developed by ObjectSpace Inc. and, although not primarily designed as a MAS, it supports both mobile agents and mobile objects.

In the following subsections, a reference framework based on the framework proposed in [30] was used for surveying the terminology and concepts, the main technical features, and the mobile agent programming models of these four MASs.

2.1. Terminology and concepts

Although the MASs analyzed and used have many common or similar aspects, some important differences exist. The common aspects (see Table 1) of these MASs are:

- All require a server generically referred to as mobile agent server that supports the main functionalities of a mobile agent, such as execution, communication and migration.
- All provide mobile agents as autonomous and active components (or mobile active objects) that can be autonomous threads or groups of threads that are independent of the implementation level.
- All provide features to support distributed applications based on the mobile agent paradigm which are independent of the complexity, richness and ease-of-use of their respective mobile agent models.

On the other hand, the main differences between these four MASs concern the following aspects:

- **Terminology.** Different names are used to address the same concepts (mobile agent, context and place): “Mobile agents” are called “aglets”, “agents”, “mobile agents”; “Contexts” (i.e. the public interfaces represent-
ing the MAS) are named differently and can be referred to by either a single or multiple interfaces; “Places” (i.e. logical locations where mobile agents run and locally interact) are only used by Grasshopper.

- **Support level** issues. There are basically two conceptual components in all MASs: (i) the Agent Execution System (AES), also called mobile agent server, is common to all the MASs but its name as well as its architecture and functionalities are different; (ii) the Agent Execution Environment (AEE), which is represented for all the MASs by the JVM, allows mobile agents and the AES to be executed in the same process.

- **Management level** issues. A set of tools (or managers) is provided to support the management of users as well as resources such as mobile agents, agent classes, places, etc. In particular, the AES manager is directly available in Aglets and Grasshopper, whereas, for the others, it could be implemented ad hoc.

- **Application level** issues. All four MASs offer support for homogeneous mobile agent-based applications. Mobile agents are programmed using Java and proprietary mobile agent programming models of the respective MAS with agent communication mainly based on proprietary primitives of that MAS. It is worth noting that Grasshopper also provides support for heterogeneous mobile agent-based applications since it is compliant with the OMG’s MASIF [24], and Aglets offers a KQML Java-based package to exchange KQML messages [19].

### 2.2. Technical features

Table 2 summarizes the main technical features of the reference MASs. The main focus is on the following aspects:

- **Naming.** The type of naming of the mobile agent server location is mainly based on URLs whereas that of the mobile agent is centered on a location-independent name which can have a string or an URL-based representation. A proxy mechanism, used to refer to remote mobile agents, is offered by all the reference MASs. The lookup of mobile agents is distributed in Ajanta, Grasshopper, and Voyager, whereas it is enabled in the same local context in Aglets.

- **Communication.** All the reference MASs provide low-level communication primitives based on message transmission and compliant with the direct coordination model. The semantics of the message transmission can be synchronous, asynchronous or based on futures. Aglets, Grasshopper and Voyager offer all three modes whereas Ajanta only provides synchronous remote method invocations.

- **Migration.** The type of migration offered by all the reference MASs is weak (or stateless) [18] with class transfer occurring mainly on-demand, even though Aglets can transfer all the classes along with the agent. Class transfer in all four MASs is supported by the codebase which can be co-located with the agent server or placed on a remote web server. The transfer protocols are ATP (Agent Transport Protocol) in Aglets and Ajanta, and are proprietary or HTTP-based in the other MASs.

- **Security.** The most secure MASs are Ajanta and Grasshopper which offer several security mechanisms for authorization, authentication, access control, privacy and integrity of the mobile agents and the agent server. Aglets and Voyager only provide basic mechanisms for specifying access rights of local and remote mobile agents.

### 2.3. Mobile agent programming models

As pointed out in Section 2.1, significant differences exist between the mobile agent programming models and the APIs of the four reference MASs. In Table 3 the basic abstractions, the lifecycle methods, and the main basic primitives are reported:

<table>
<thead>
<tr>
<th><strong>Term</strong></th>
<th><strong>Aglets</strong></th>
<th><strong>Ajanta</strong></th>
<th><strong>Grasshopper</strong></th>
<th><strong>Voyager</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reference Model</strong></td>
<td><strong>Mobile Agent</strong></td>
<td>Aglet</td>
<td>Agent</td>
<td>Mobile/Stationary Agent</td>
</tr>
<tr>
<td><strong>Context</strong></td>
<td><strong>Agent Environment</strong></td>
<td>Agent Environment</td>
<td>Agent System</td>
<td>multiple interfaces</td>
</tr>
<tr>
<td><strong>Place</strong></td>
<td>none</td>
<td>none</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td><strong>Support Level</strong></td>
<td>AES</td>
<td>Agletsd+Tahiti</td>
<td>AjantaServer</td>
<td>Agency</td>
</tr>
<tr>
<td><strong>AEE</strong></td>
<td>JVM</td>
<td>JVM</td>
<td>JVM</td>
<td>JVM</td>
</tr>
<tr>
<td><strong>Management Level</strong></td>
<td>AES Manager</td>
<td>Tahiti</td>
<td>none</td>
<td>Agency GUI</td>
</tr>
<tr>
<td><strong>Application Level</strong></td>
<td><strong>Agent Communication</strong></td>
<td>Java+J-API</td>
<td>Java</td>
<td>Java+CORBA</td>
</tr>
<tr>
<td><strong>Agent Programming Language</strong></td>
<td>Java+Aglets Programming Model</td>
<td>Java+Ajanta Programming Model</td>
<td>Java+Grasshopper Programming Model</td>
<td>Java+Voyager Programming Model</td>
</tr>
</tbody>
</table>
Table 2
Main technical features of the reference MASs

<table>
<thead>
<tr>
<th>REFERENCE MODEL</th>
<th>AGLETS</th>
<th>AJANTA</th>
<th>GRASSHOPPER</th>
<th>VOYAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMING</td>
<td>TYPE</td>
<td>URLS based on DNS names</td>
<td>Location-independent global names</td>
<td>Location-dependent global ID, as well as local proxy</td>
</tr>
<tr>
<td></td>
<td>AGENT SERVER LOCATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;hostname&gt;:&lt;port&gt;/&lt;context&gt;</td>
<td>urn:ans:&lt;hostname&gt;/&lt;username&gt;/sername&gt;</td>
<td>&lt;protocol&gt;:&lt;host&gt;:&lt;port&gt;/&lt;agency&gt;/&lt;place&gt;</td>
<td>&lt;hostname&gt;:&lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>AGENT ID</td>
<td>AgentID &lt;&lt;class&gt;&gt;</td>
<td>urn:ans:&lt;hostname&gt;/&lt;username&gt;/sername/&lt;agentname&gt;</td>
<td>(String representation)</td>
</tr>
<tr>
<td></td>
<td>PROXY</td>
<td>AgentProxy &lt;&lt;class&gt;&gt; Remote Interface/RMI Stub</td>
<td>Distributed enabled by the Name Registry</td>
<td>Remote-accessible Local Namespace</td>
</tr>
<tr>
<td></td>
<td>LOOK_UP</td>
<td>Local in the AgletContext</td>
<td>Distributed enabled by the Region Registry</td>
<td></td>
</tr>
<tr>
<td>SEMANTICS</td>
<td>Type</td>
<td>WEAK</td>
<td>WEAK</td>
<td>WEAK</td>
</tr>
<tr>
<td></td>
<td>CLASS TRANSFER</td>
<td>On-demand/Bundle</td>
<td>On-demand</td>
<td>On-demand</td>
</tr>
<tr>
<td>CODEBASE</td>
<td>CODEBASE</td>
<td>ATMP Manager</td>
<td>Ajanta Server</td>
<td>Voyager-enabled HTTP Server</td>
</tr>
<tr>
<td></td>
<td>&lt;host&gt;:&lt;port&gt;</td>
<td></td>
<td></td>
<td>http://&lt;voyagerservername&gt;</td>
</tr>
<tr>
<td>AUTHORIZATION AND ACCESS CONTROL</td>
<td>Statically specified access rights, based on two security levels: Trusted and Untrusted</td>
<td>Credentials-based resource access. Authorization based on agent’s owner.</td>
<td>Access control policies based on authentication/authorization</td>
<td>Statically specified access rights, based on two security levels: Native and Foreign</td>
</tr>
<tr>
<td>AGENT PRIVACY AND INTEGRITY</td>
<td>none</td>
<td>DES / EC/ESM protocol. Detection of tampering of agent state and code.</td>
<td>SSL (Secure Socket Layer) / MAC (Message Authentication Codes)</td>
<td>none</td>
</tr>
<tr>
<td>AGENT AND SERVER AUTHENTICATION</td>
<td>none</td>
<td>Public-key Cryptography</td>
<td>X.509 certificates</td>
<td>none</td>
</tr>
<tr>
<td>METERING, CHARGING, AND PAYMENT MECHANISMS</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

- Basic abstractions. While in Aglets, Ajanta and Grasshopper a mobile agent is defined as a class which extends the basic agent class, in Voyager an agent can be defined using either the Agent class or any class as any object can be transformed into an agent at run-time. The context can be accessed by a single direct reference of a specific type in Aglets, Ajanta and Grasshopper, whereas in Voyager context is accessed through static methods of several classes.

- Lifecycle. The lifecycle model is different for all the reference MASs. The Aglets and Grasshopper lifecycles are based on an event model according to which an agent is notified through callbacks if specific events occur: (i) after an agent creation the onCreation method in Aglets and the init method in Grasshopper are invoked; (ii) before and after an agent migration the onDispatching and the onArrival methods in Aglets and the beforeMove and the afterMove methods in Grasshopper are respectively invoked. The main activity of an agent is performed in the run method in Aglets and in the live method in Grasshopper. The Ajanta lifecycle is based on a chaining model in which, after agent creation, the arrive, run and departure methods are sequentially invoked. The Voyager lifecycle is event-based if and only if the IMobile interface is implemented. However, the main activity of a Voyager mobile agent can be performed in any method.

- Main primitives. The main primitives of MASs are those which allow an agent to create new agents, migrate to a destination agent server, communicate with other agents, and lookup local as well as remote agents. Although these main primitives are provided by the API of every MAS, they are obviously different not only with respect to the primitive signature but also with respect to the model on which the primitive is based. While creation and migration primitives of the reference MASs are all based on a similar model, in some cases, the models on which the communication and lookup primitives are based are significantly different, as shown in Table 3.

3. The Interoperable Mobile Agent model

An Interoperable Mobile Agent (IMA) is represented by the pair <HLP, LLP>, where:

- LLP is the Low-Level Part, which depends on the specific MAS where the mobile agent is operating and changes after a heterogeneous migration, assuming the “shape” of the mobile agent of the new hosting MAS.
- HLP is the MAS-neutral High-Level Part, which does not change during the mobile agent lifecycle. HLP is centered
on an event-driven lightweight mobile agent model and uses a proxy-based communication infrastructure. HLP is adapted to LLP through an interface offering basic services to the HLP which are implemented by the LLP.

In the next subsections we describe the mobile agent model on which the HLP of an IMA is based, the architectural model of the communication infrastructure among IMAs and the bridging architecture model which allows an IMA to cross from one heterogeneous MAS to another.

### 3.1. The event-driven lightweight mobile agent model

An event-driven lightweight mobile agent is represented by the tuple \(<Id, S, \text{Beh}, Tc, \text{Eq} \rangle\), where Id is the unique agent identifier, S is the agent state, Beh is the agent behaviour, Tc is the thread of control associated to the agent, Eq is the queue of the events received and to be handled by the agent. An event is defined by the tuple \(\text{EVT} = \langle \text{Source, Target, Type, Args} \rangle\), where Source and Target are respectively the source and the target of the event, Type represents the type of event and Args is a list of parameters which the event can contain.

An event executes the Event Processing Cycle (EPC) (see Fig. 2a) which consists of the cyclically dequeuing and processing of an event according to its state and behaviour. During event processing, the agent can perform one of the following actions:

- Computations which can directly affect the agent state, creations of agents by invoking the primitive \textit{create}, and/or transmissions of events by means of the primitive \textit{send}.
- Migration by using the primitive \textit{migrate}.

An event processed by an agent can be generated either by the agent itself (self-event) or by another agent (external event). While the generation of self-events is a proactive mechanism by which an agent is able to autonomously drive its activity, external events are a means of interaction. At the end of the event processing, the agent is either in a new, fully defined active state (i.e. ready to react to the next event) or in the transit state, in which the agent remains until migration is completed. The migration is asynchronous in that it is requested by means of the transmission of the Move self-event, which is queued at the head of the EQ and, when processed, induces the actual agent migration \([9,10]\).

The event handling and state evolution of an example agent are shown in Fig. 2b and c. The example agent is able to react to the events Move, Reply, and Terminate. In particular, the handling of the Reply event triggers the execution of the following sequence of actions: invocation of the compute method which performs computations affecting the internal state of the agent, creation of agent Y, transmission of the EvtX event to a target agent, and transmission of the Move event to itself. After this execution, the agent is in the state \(X_{eq}\), in which it receives the Move event that actually triggers agent migration to the target destination.

### 3.2. The proxy-based communication infrastructure

Inter-agent communication relies on a message-based coordination model which allows for asynchronous transmission and reception of events between mobile agents. The communication infrastructure is based on Proxies which ensure agent location transparency. A mobile agent is always associated to its Proxy: \(<\text{MA}, P>\), where MA is the Mobile Agent and P is the Proxy of MA. The Proxy P is composed of N components: \(P = <P_1, \ldots, P_N>\), one per agent server that the mobile agent has visited. The component \(P_i\) is a tuple \(<\text{Id}, \text{FW, MBX}>\), where \text{Id} is the identifier of the Proxy component, FW is the forwarder, i.e., the
Id of the chained Proxy component, and MBX is the associated mailbox, which buffers events during the migration of the MA.

In order to send an event to a partner, an MA should know at least the Id of one of the Proxy components of the partner MA. Upon the reception of an event, the Proxy component inserts the event either into the event queue of the MA, if the MA is present, or in the MBX, if the FW is not active; otherwise, the event is forwarded to the chained Proxy component (see Fig. 3).

Once the MA completes its migration, its proxy component recovers the events which may be archived in the MBX. The delivery policy of the events can be programmed as follows:

1. **Sync MBX retrieval.** The proxy component synchronously gets the MBX from the previous chained proxy component. FW is activated as soon as the getMBX request is received. MA is started once the MBX has been received (see Fig. 4a).

2. **Async MBX retrieval.** The proxy component requests the MBX of the previous chained proxy component. FW is activated as soon as the requestMBX is received. MBX is then transmitted “all together” to the requesting chained proxy component. MA is started after the MBX has been requested (see Fig. 4b).

3. **Active MBX forwarding.** The proxy component activates the FW of the previous chained proxy component. FW is activated as soon as the forward request is received. Afterwards, the MBX is transmitted “event by event” to the requesting chained proxy component. MA is started after the forward request has been transmitted (see Fig. 4c).

### 3.3. Bridging mobile agents

Within a network of homogeneous MASs the migration of the IMA is fully supported by its LLP. Conversely, in order to move between two heterogeneous MASs, the IMA makes use of mobile agent bridges (MABs). A MAB is a reactive software infrastructure which is able to receive an IMA from one MAS and inject it into the target heterogeneous MAS. MABs are to mobile agents as LAN bridges are to network packets. However, LAN bridges are active while MABs are reactive, i.e. they provide the bridging service upon request of the IMA.

Let \(IMA = \langle HLP, LLP_{SOURCE} \rangle\) be an interoperable mobile agent that wishes to migrate from a SOURCE MAS of the X type to a TARGET MAS of the Y type, where \(X \neq Y\); the migration schema is as follows (Fig. 5):

---

**EPC:**

```java
while (!terminated) {
  e = dequeue(EQ)
  handleEvent(e)
}
```

**EVT**

```java
handleEvent(e:EVT) {  
  if (Type(e)=='Move')  
    migrate(Destination(e))
  else if (Type(e)=='Reply') {
    compute()
    create(Ys)
    send(new EVT(company))
    send(new Move(self()))
  }  
  else if (Type(e)=='Terminate')
    terminated=true;
}
```

**Fig. 2.** (a) The Event Processing Cycle; (b) event handling method of an example agent in pseudo-code; (c) evolution of the state and basic actions of the example agent.
IMA contacts (step 1) a MAB (MAB\textsubscript{X $\rightarrow$ Y}), which provides the bridging from X to Y, by passing its HLP along with the identifier of the TARGET MAS.

After receiving the HLP, the MAB\textsubscript{X $\rightarrow$ Y} creates (step 2) a new LLP specific to Y (LLP\textsubscript{TARGET}), adapts (step 3) HLP to LLP\textsubscript{TARGET} so obtaining the IMA* = \langle HLP, LLP\textsubscript{TARGET} \rangle, and homogeneously migrates (step 4) the IMA* to the TARGET MAS.

Although the LLP is substituted, the HLP, which contains the activity of the mobile agent, is preserved, making the bridging process completely transparent.

4. The java-based interoperable mobile agent framework

The Java-based Interoperable Mobile Agent Framework (JIMAF) is designed and implemented to hide heterogeneity of Java-based MASs and can actually be layered atop available Java-based MASs without requiring any modifications of the underlying MASs; to date JIMAF has been successfully layered atop Aglets, Ajanta, Grasshopper and Voyager.

In the following subsections, we describe the structure and the main dynamics of JIMAF and use an example to illustrate the main features of JIMAF.

4.1. Structure and dynamics

The structure of JIMAF consists of three software layers (see Fig. 6):

- The Interoperable Mobile Agent Layer (IMAL), which implements the HLP of the IMA according to the event-driven mobile agent model and the Proxy-based communication infrastructure (see Section 3). In particular, the Proxy-based communication infrastructure is implemented through the Java RMI middleware. IMAL is composed of: (i) the InteroperableAgent class, which is the abstract class implementing the basic event-driven mobile agent, and the WrapperAgent and BridgeAgent classes which extend the InteroperableAgent class and support the implementation of the wrapping and bridging mechanisms; (ii) the AgentDescriptor class, which encloses the agent identifier as well as the agent location which is represented by the Location class; (iii) the Event and EventQueue classes, which are the classes representing events and event queues, respectively; (iv) the AgentProxy and EventForwarder classes, which are the classes implementing agent proxy components and event forwarders. An AgentProxy is an RMI-enabled remote object.

- The Adaptation Layer (AL), which is an adapting interface (IAgent) declaring the following set of methods used by the HLP and implemented by the LLP of the IMA: (i) create, which creates a new IMA; (ii) migrate, which uses the platform-dependent migration primitive to migrate an IMA; (iii) send, which uses a platform-dependent transmission primitive to send an event to a target IMA with homogeneous LLP; (iv) lookup, which uses the platform-dependent lookup service to search for an IMA with homogeneous LLP.

- The Platform-dependent Mobile Agent Layer (PMAL), which consists of an extendible set of mobile agent classes representing the different LLPs of an IMA. Currently the available set consists of the AgletsAgent, AjantaAgent, GrasshopperAgent, and VoyagerAgent classes which extend the base classes of their respective reference MAS (see Section 2.3).

In the following subsections the structure and the main dynamics of JIMAF are presented.
4.1.1. Lifecycle management

The creation of an IMA, that can be carried out by a server application or by another IMA by using the create method, consists of the following steps:

1. Creation of an AgentDescriptor that contains the unique agent identifier that is independent of the LLP of an IMA and never changes during its lifecycle. The string-based structure of the identifier is: `<HostName> - <UserName> - <AgentServerName> - <AgentName> - <Port>`, where: `<HostName>` is the name of the host where the IMA was created, `<AgentServerName>` and `<Port>` are, respectively, the name of the agent server and the port number of the LLP where the IMA was created, `<AgentName>` is the name of the agent, and `<UserName>` is the name of the user owning the agent. For example, `pards1.deis.unical.it-gfortino-server1-aja1-4095` is the agent identifier of an IMA created atop Ajanta.

2. Creation of the HLP of the IMA by passing the created AgentDescriptor to the constructor of the InteroperableAgent class.
3. Creation of the LLP by passing the created HLP to the LLP constructor; this operation depends on the MAS API of the LLP (e.g., see the Creation primitives in Table 3, Section 2.3).

4. Invocation, carried out by the LLP, of the method setIAgent (see Fig. 6) of the InteroperableAgent class to adapt the HLP to the LLP.

The lifecycle of an IMA (see Fig. 7), which goes from its creation to termination and is driven by its LLP in the context of which it is always executed (i.e. the LLP lends its thread to the HLP), is enclosed in the onArrival, run and onDeparture methods of the HLP which are invoked by the LLP in the following order:

1. **onArrival** is invoked just after the IMA is created and every time it arrives at a new agent server. This contains the code for creating the proxy component and for binding or rebinding the IMA to such a proxy component.

2. **run**, which contains the basic code implementing the EPC (see Section 3.1). The processing of events is carried out through the handleEvent method (see Fig. 6). The EPC is exited after processing the “migrate” or the “terminate” event.

3. **onDeparture**, which contains the code for handling the migration process of the IMA. If no migration is carried out during the execution of this method, the IMA terminates.

### 4.1.2. Communication between interoperable agents

Communication between IMAs is based on events as described in Section 3.2. In particular, to communicate with a partner, an IMA must know the URL of a Proxy component of the partner. This information is included in the AgentDescriptor of the agent partner. The URL format is as follows: rmi://<HostName>:<Port>/<AgentID>, where: <HostName> and <Port> are respectively the name of the host and the port where the Proxy component is located, and <AgentID> is the identifier of the IMA.

The following steps are carried out by an IMA to transmit an external event to a target IMA or a self-event:

1. Creation of the event by using the URL of its current proxy component (URLSender), the URL of a proxy component of the receiving agent (URLReceiver), the event type (tag), and the list of the event parameters (args).

2. Invocation of the send method (see Fig. 6) by passing the created event as parameter.

The send method operates as follows: if URLSender is equal to URLReceiver, the event is directly inserted into the EQ of the transmitting agent itself; otherwise, the AgentProxy component of the receiving agent is looked up by using the RMI Naming lookup method and the send method of the looked-up AgentProxy component is invoked by passing the event as a parameter. The AgentProxy component of the receiving IMA passes the event to the agent by invoking the dispatch method (see Fig. 6) if the agent is present; otherwise, it queues the event into the agent mailbox or activates an EventForwarder component to forward it. Fig. 8 shows an UML interaction diagram describing all the steps involved in the transmission of an event from the interoperable mobile agent IAₓ to the migrating remote interoperable mobile agent IAᵧ which, after its migration, is bound to the AgentProxy Py₂ which is in turn linked to the AgentProxy Py₁.

Interactions between IMAs with homogeneous LLPs (i.e. operating on MASs of the same type) can also exploit the specific communication mechanisms of the hosting platforms through the lookup and send methods defined in the IAgent interface (see Fig. 6).
4.1.3. WrapperAgents enabling communication between interoperable agents and platform-dependent agents

To enable communication between interoperable agents and platform-dependent agents, so allowing for communication interoperability between IMAs and platform-dependent agents, wrapping techniques can be exploited [11]. In particular, JIMAF exploits the Wrapper Agents (WAs), special JIMAF-based agents which wrap platform-dependent agents and act as mediators capable of interacting both with IMAs by using JIMAF-based events and with platform-dependent agents by using platform-dependent communication mechanisms. This allows for the use/reuse of platform-dependent code and functionalities in the context of a JIMAF-based application which is thus constituted by both JIMAF-based agents (IMAs and WAs) and platform-dependent agents.

A JIMAF-based Wrapper Agent is a stationary IMA whose HLP is constituted by a class which extends the WrapperAgent abstract class (see Fig. 6) and whose LLP depends on the hosting platform. In particular, the WrapperAgent must be extended by overriding:

- the translate method to support the adaptation of the communication content from JIMAF-based events to platform-dependent formats;
- the constructReply method to support the adaptation of the communication content from platform-dependent formats to JIMAF-based events;
- the handleEvent method to handle the JIMAF-based events sent by an IMA to request services offered by platform-dependent agents.

Fig. 9 shows a general interaction between an IMA and a platform-dependent agent (PMA), mediated by a WA, in which the interaction initiator is the IMA. In particular, the IMA sends its request (evtReq) to the WA that, in turn, translates its content into a platform-dependent format through the translate method and incorporates this content into a platform-dependent request which is sent to the PMA. As soon as the WA obtains the reply from the PMA, it constructs the reply event (evtRep) through the constructReply method and sends this event to the IMA.

4.1.4. Bridge agents: enabling heterogeneous migration

Heterogeneous migration is enabled by MABs (Mobile Agent Bridges), which are currently implemented as stationary BridgeAgents that extend the InteroperableAgent. To enable migration between MASs made interoperable, at least one MAB for each MAS must be developed and deployed. An IMA with the LLP of type X, which is going to migrate to an agent server of type Y (X ≠ Y), can send an event of migration request (migrRequest) to any MAB of type Y. A migrRequest contains the HLP of the IMA along with the Location of the target agent server. The contacted MAB therefore creates a new LLP, adapts the HLP of the incoming IMA to it and performs the homogeneous migration of the obtained IMA towards the target agent server (see Section 3.3).

4.2. Programming an application through JIMAF

The main features of JIMAF are illustrated by describing a JIMAF-based solution for retrieving information from remote data repositories which are managed by agents operating on heterogeneous agent platforms. This problem could be effectively approached through the mobile agent paradigm by employing a mobile agent which migrates to the locations of data

![Fig. 8. Sequence diagram of the transmission of an event to a migrating interoperable mobile agent.](image)
repositories to retrieve requested data by interacting with the agents which manage these data repositories. As demonstrated in the literature [28], such a solution is particularly effective for reducing the time necessary to complete the distributed information retrieval task. However, if data repositories are managed by agents operating on heterogeneous agent platforms, the exploitation of this solution requires that the mobile agent performing the information retrieval task is capable of migrating among heterogeneous agent platforms, executing on these platforms and interacting with the platform-dependent agents developed and deployed for the management of data repositories. As discussed in Section 6, to date, JIMAF is the only high-level approach which provides full support for the programming of such a roaming mobile agent which satisfies all the aforementioned requirements.

The JIMAF-based solution, which exploits the already available platform-dependent local agents (DataAgents) managing the data repositories (in Fig. 10 such an agent for an Aglet platform is shown), relies on the definition and implementation of:

- an IMA (RoamingDataAccessor) which, according to a given itinerary, roams through the heterogeneous agent platforms where the data repositories are managed by the DataAgents, retrieves the desired data and departs for the next location;
- a set of local wrapper agents (DataWrapperAgents), one for each different type of platform, which may be included in the itinerary. Each DataWrapperAgent mediates the communication between the RoamingDataAccessor and a local DataAgent by using the wrapping techniques illustrated in Section 4.1.3. In particular, at each location of the itinerary, the RoamingDataAccessor interacts through JIMAF-based events with the local DataWrapperAgent, which, in turn, interacts with the local DataAgent through the interaction methods provided by the specific platform.

In the following subsections, the utilization of the most important JIMAF features are highlighted by describing the programming of the RoamingDataAccessor and a DataWrapperAgent. The complete code of the JIMAF-based application can be requested from the JIMAF website [17].

4.2.1. Programming the RoamingDataAccessor

The behaviour of the RoamingDataAccessor was specified through the definition of its HLP (see Section 4.1.1)

```java
package dataagentagens;
import com.ibm.agl.*;
import java.sql.*;
import java.sql.ResultSet*;
import java.util.*;
import java.util.*;

public class AgletsDataAgent extends Aglet {
  String JDBCDriver, DB_URL;
  public void onCreate(_packet* args) {
    Vector v = (Vector) args;
    JDBCDriver = (String) v.remove(0);
    DB_URL = (String) v.remove(0);
  }
  public void run() {
    public void handleMessage(Message msg) {
      if (msg很大程度?DataRequest) {
        String QUERY = (String) msg.getArg();
        try {
          Class.forName(JDBCDriver);
          connection = DriverManager.getConnection(DB_URL);
          Statement statement = connection.createStatement();
          ResultSet resultSet = statement.executeQuery(QUERY);
          CachedRowSetImpl serializableResultSet = new CachedRowSetImpl();
          serializableResultSet.populate(resultSet);
          msg.sendReply(serializableResultSet);
        } catch (Exception e) {
          System.out.println("Exception: " + e.toString());
        } finally {
          try {
            if (connection != null) connection.close();
          } catch (SQLException e) {}
          return true;
        } else return false;
      }
    }
  }
}
```

Fig. 10. The AgletsDataAgent class.
whereas the LLP, which depends on the agent platforms which host the *RoamingDataAccessor* during its roaming task, is already provided by JIMAF for the platforms which have been made interoperable (see Section 4.1). In particular, the definition of the HLP, whose code is shown in Fig. 11, was carried out by overriding:

- the *onArrival* method to specify the behaviour of the agent after its creation or migration. In particular, after its creation, the agent migrates onto the first platform of the assigned itinerary and after each subsequent migration the agent constructs an event to be sent to the *DataWrapperAgent* which is active on the current hosting platform; this event contains the URL of the proxy of the *RoamingDataAccessor* (*URLSender*), the event type (*DataRequest*), and the event content (*evtArgs*) related to the query for retrieving the desired data. When the agent completes its itinerary and returns to the starting location (*startingLocation*), it analyzes the gathered data through the *analyseData* method.

- the *handleEvent* method to specify how the events that can be received by the agent are to be handled. In particular: (i) to handle the *DataReply* event, the agent adds the data extracted from the content of the event which is sent by a *DataWrapperAgent* to its gathered data; in addition, if the itinerary is not fulfilled, the agent generates a migrate event with itself as target (*URLSender = URLReceiver*) and the next location of the itinerary as content, otherwise, the agent generates and sends a Finish event to itself; (ii) to handle the Finish event the agent generates a migrate event for triggering its migration to the *startingLocation* on which the gathered data will be analysed.

The following relevant JIMAF features are highlighted from the given description of the *RemoteDataAccessor*: (i) event-based programming of IMAs which allow for the proactive driving of the IMA behaviour (see migrate and finish events) and communication with other IMAs (see *DataRequest* and *DataReply* events); (ii) transparent programming of heterogeneous migrations through the generation of events of the migrate type which are automatically managed by the JIMAF-based bridge infrastructure.

### 4.2.2. Programming the DataWrapperAgents

For each type of platform on which the *RoamingDataAccessor* can migrate, a *DataWrapperAgent*, which adapts the communication mechanism of JIMAF to that of the specific type of platform, must be defined (see Section 4.1.3). Each specific *DataWrapperAgent* implements: (i) the *translate* and *constructReply* methods to support the translation of the communication content from/to JIMAF-based events to/from platform-dependent formats; (ii) the *handleEvent* method to handle the communication requests sent by the *RoamingDataAccessor* (see Section 4.2.1). Fig. 12 shows the extension of the *DataWrapper-Agent* class for the Aglets platform (*AgletsDataWrapper-Agent*); similar extensions for the Ajanta, Grasshopper, and Voyager were also developed by exploiting the interaction mechanisms specific for such platforms (see Section 2.2).

In the following, the above mentioned methods of the *AgletsDataWrapperAgent* class (see Fig. 12) are described:

- the *translate* method extracts the event content (i.e. the query for data retrieving) and returns an Aglets-specific *Message*;

```java
package jimar.demo.remotedataaccessor;
import jimar.interop.interoperableAgent;
import java.sql.*;
import javax.sql.rowset.*;
import java.util.*;

public class RoamingDataAccessor extends InteroperableAgent {
    private int locationIDNumber = 0;
    private boolean roamingStarted=false;
    private Location startingLocation;
    private String itinerary, dataWrapperAgents, data;
    String URLSender, URLReceiver, evTag, query;
    Object evArgs;
    public RoamingDataAccessor-AgentDescriptor ad, Vector itinerary,
            Vector dataWrapperAgents; String query;
    super(ad);
    this.itinerary=itinerary; locationsToVisit = itinerary.size();
    this.query=query; this.data=data; new Vector(locationsToVisit);
}

public void onArrival{
    super onArrival();
    if(roamingStarted=false) {
        startingLocation = getinfo().getCurrentLocation();
        if (locationsToVisit.size()==0) {
            roamingStarted=true;
            URLSender = myInfo.getName();
            URLReceiver = myInfo.getName();
            evTag = "migrate";
            evArgs = (Location)itinerary.elementAt(locationIDNumber); send(new Event(URLSender, URLReceiver, evTag, evArgs));
        }
    }else if(roamingComplete=false) {
        URLSender = myInfo.getProxyURL();
        AgentDescriptor AgReceiver = (AgentDescriptor)
        (dataWrapperAgents.elementAt(locationIDNumber));
        URLReceiver = AgReceiver.getProxyURL();
        evTag = "DataRequest";
        evArgs = query;
        send(new Event(URLSender, URLReceiver, evTag, evArgs));
        if (analyseData();
        }
    }else {
        URLSender = myInfo.getName();
        URLReceiver = myInfo.getName();
        evTag = "migrate";
        evArgs = (Location)itinerary.elementAt(locationIDNumber); send(new Event(URLSender, URLReceiver, evTag, evArgs));
    }
    super.handleEvent();
}

public void onDeparture() {
    super.onDeparture();
    private Object filter (Object data) { return data; }
    private void analyseData { (...);} 
}
```

Fig. 11. The *RoamingDataAccessor* class.
- the ConstructReply method generates a DataReply event with the content extracted from the data which has been provided by the AgletsDataAgent (see Fig. 10);
- the handleEvent method handles the DataRequest event by invoking the translate method so obtaining an Aglets-specific Message (msg), transmitting it to the AgletsDataAgent (through the synchronous sendMessage method of Aglets), constructing the DataReply event (evtRep) through the constructReply method, and sending it to the RemoteDataAccessor.

The interaction between the RemoteDataAccessor and the AgletsDataAgent mediated by the AgletsDataWrapperAgent is shown in Fig. 13.

The following relevant JIMAF features are highlighted from the given description of the AgletsDataWrapperAgent:

(i) communication between interoperable mobile agents and platform-dependent agents (RoamingDataAccessor and AgletsDataAgent) mediated by wrapper agents (AgletsDataWrapperAgent) to request services offered by platform-dependent agents (see Section 4.1.3); (ii) use/reuse of platform-dependent code and functionalities in the context of a JIMAF-based application which is thus constituted by interoperable mobile agents (RoamingDataAccessor), wrapper agents (e.g. AgletsDataWrapperAgent) and platform-dependent agents (e.g. AgletsDataAgent).

5. Performance evaluation

To evaluate the performance of JIMAF for the currently exploited MASs and quantify the performance overhead which a high-level approach inevitably introduces, several micro benchmarks and a micro kernel were implemented and executed to measure performance indexes defined ad hoc. In particular, the micro benchmarks Ping-Pong, Lookup, Message-Transmission, and Creation were used to test the performance of simple activities implemented by the basic elements of an agent platform. The micro kernel Roam was chosen from the different micro kernels which have been proposed in the literature [8] for testing the important JIMAF feature of heterogeneous migration.

The experimentation phase was carried out on a local testbed consisting of a set of high performance PCs connected through a switched Fast Ethernet. Moreover, the software environment was enabled by Windows XP Professional and the Java 2 Standard Edition v.1.4.1.

5.1. Micro Benchmarks

The implemented micro benchmarks Ping-Pong, Lookup, Message-Transmission, and Creation respectively capture the overhead of agent migration, agent lookup, message transmission between two agents, and local agent creation. In particular, the defined micro benchmarks allow for the evaluation of the following performance indexes (see Table 4 for notation):

![Fig. 12. The AgletsDataWrapperAgent class.](image)

![Fig. 13. The UML Sequence diagram of the example application w.r.t. Aglets.](image)
Table 4

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>A set of Agent Platforms</td>
</tr>
<tr>
<td>p ∈ AP</td>
<td>An Agent Platform</td>
</tr>
<tr>
<td>H&lt;sup&gt;p&lt;/sup&gt;</td>
<td>A host running an instance of p</td>
</tr>
<tr>
<td>IA&lt;sup&gt;p&lt;/sup&gt;</td>
<td>An Interoperable Agent running on an instance of p</td>
</tr>
<tr>
<td>NA&lt;sup&gt;p&lt;/sup&gt;</td>
<td>A Native Agent programmed using the API of the platform p and running on an instance of p</td>
</tr>
</tbody>
</table>

1. **PPT (Ping-Pong Time)**, which indicates the time required by **IA<sup>p</sup>** (**NA<sup>p</sup>**) to migrate from **H<sub>x</sub></sup> to **H<sub>y</sub></sup>** and return to **H<sub>y</sub></sup>. PPT was calculated by executing the **Ping-Pong** benchmark for the following three scenarios:
   - **(PP1) Ping-Pong with Caching and Local Class Loading**: the code of **NA<sup>p</sup>** and **IA<sup>p</sup>** is located on both **H<sub>x</sub></sup> and **H<sub>y</sub></sup>, and the measurements are acquired without rebooting <sup>p</sup> on **H<sub>x</sub></sup> and **H<sub>y</sub></sup>.  
   - **(PP2) Ping-Pong with No Caching and Local Class Loading**: the code of **NA<sup>p</sup>** and **IA<sup>p</sup>** is located on both **H<sub>x</sub></sup> and **H<sub>y</sub></sup>, and the measurements are acquired by rebooting <sup>p</sup> on <sup>p</sup> for each trial to avoid class caching effects.
   - **(PP3) Ping-Pong with No Caching and Remote Class Loading**: the code of **NA<sup>p</sup>** and **IA<sup>p</sup>** is located only on **H<sub>z</sub></sup>, and the measurements are acquired by rebooting <sup>p</sup> on **H<sub>x</sub></sup> and **H<sub>y</sub></sup> for each trial to avoid class caching effects.

2. **LKT (LookUp Time)**, which represents the time that **IA<sup>p</sup>** (**NA<sup>p</sup>**), created on **H<sub>y</sub></sup>, takes for determining the location of another agent **IA<sup>p</sup>** (**NA<sup>p</sup>**), created on **H<sub>z</sub></sup> and migrated from **H<sub>x</sub></sup> to **H<sub>y</sub></sup> and from **H<sub>y</sub></sup>, with the assumption that **IA<sup>p</sup>** (**NA<sup>p</sup>**) only knows the first location **H<sub>y</sub></sup> of **IA<sup>p</sup>** (**NA<sup>p</sup>**).

3. **MTT (Message Transmission Time)**, which is the time between the transmission of a message from **IA<sup>p</sup>** to **IA<sup>p</sup>** (**NA<sup>p</sup>**), hosted respectively on **H<sub>x</sub></sup> and **H<sub>y</sub></sup>, and reception of the acknowledgement, sent from **IA<sup>p</sup>** (**NA<sup>p</sup>**). **MTT** was calculated in the following two scenarios by the execution of the **Message-Transmission** benchmark:
   - **(MT1) One-Way Message Transmission**: **IA<sup>p</sup>** (**NA<sup>p</sup>**) simply sends a message to **IA<sup>p</sup>** (**NA<sup>p</sup>**).
   - **(MT2) Service Request**: **IA<sup>p</sup>** (**NA<sup>p</sup>**) requests an **echo service** from **IA<sup>p</sup>** (**NA<sup>p</sup>**), which consists of printing a received string on its shell and returning the string.

4. **CRT (Creation Time)**, which represents the time which is taken for the local creation of **IA<sup>p</sup>** (**NA<sup>p</sup>**).

Some results obtained by executing the micro benchmarks are reported in **Tables 5 and 6** which show the most significant outcomes regarding: (i) the maximum performance overhead **OH%** of the IA for the defined performance indexes and the platform on which such overhead was measured (**Table 5**); (ii) the maximum and minimum average times of the IA for the defined performance indexes and the platform on which such times were measured (**Table 6**).

In particular, **OH%** is calculated as 
\[
\frac{\text{PIX}_{IA}}{\text{PIX}_{NA}} - 1 \times 100,
\]
where **PIX<sub>IA</sub>** is the performance index related to **IA<sup>p</sup>** while **PIX<sub>NA</sub>** is the same performance index related to **NA<sup>p</sup>**. In addition, each performance index was obtained by averaging over a sample of 50 measurements.

The analysis of the results shows that, although the overhead is higher for the highest performance platform, this inevitable overhead is nonetheless contained. Moreover, for each performance index, the performance of the IA is comparable with the performance of the lowest performance platform.

In particular:

- The maximum **OH%** for the PPT performance index was obtained on Aglets (see **Table 5**) due to its high performance migration mechanism. Moreover, the best performance for the PPT (**PP3**) index was obtained on Aglets, whereas the worst performance was obtained on Grasshopper (see **Table 6**), which is the lowest performance platform with respect to agent migration.
- The maximum **OH%** for the LKT performance index was obtained on Ajanta (see **Table 5**) due to the very fast registry-based lookup mechanism of the Ajanta system. Moreover, since the lookup mechanism of the IA is independent of the platforms on which the IA operates, the LKT performance index calculated was the same for all the platforms (see **Table 6**) and comparable with the lookup performance of the lowest performance platform (Voyager).
- The maximum **OH%** for the MTT (**MT2**) performance index was obtained on Voyager (see **Table 5**) which is the system providing the highest performance mecha-
nisms for message transmission. Moreover, since the message transmission mechanism of the IA is independent of the platforms on which it operates, the MTT (MT2) performance index calculated is the same for all the platforms (see Table 6) and comparable with the message transmission performance of the lowest performance platform (Ajanta).

- The maximum OH% for the CRT performance index was obtained on Voyager (see Table 5) due to its high performance mechanism for agent creation which adopts thread pooling to improve agent creation. Moreover, the best performance for the CRT index was obtained on Voyager whereas the worst performance was obtained on Ajanta (see Table 6) which is the lowest performance platform with respect to agent migration since Ajanta introduces additional security data structures in the agent at creation time and registers a newly created agent into a name registry server.

5.2. Micro Kernel

The implemented micro kernel Roam allows for the evaluation of the following performance index:

- RMT (Roaming Time). Given the agent platforms \( p_1, \ldots, p_n \), running respectively on \( H^1 \), \( H^2 \), \( \ldots, H^n \), and, \( IA^{p_i} \) created on \( p_i \) (with \( 1 \leq i \leq n \)), RMT is defined as the time that \( IA^{p_i} \) takes for returning to the starting host \( H^j \) after visiting only once, all the other hosts \( H^m \) (with \( 1 \leq j \leq n \) and \( j \neq i \)); this path is defined as Completely Heterogeneous Itinerary (CHI). Given a CHI consisting of \( n \) hosts, the RMT for a number of \( CHI \equal k \) is defined as the time that \( IA^{p_i} \) takes for repeating the \( CHI \) \( k \) times.

The Roam micro kernel was executed with \( p_1 = \) Ajanta, \( p_2 = \) Aglets, \( p_3 = \) Grasshopper, \( p_4 = \) Voyager and for \( k = \{1, 2, 10, 50, 100, 500, 1000\} \). Fig. 14 shows the average RMT obtained by varying \( k \) and calculated by averaging over a sample of 50 measurements. It is worth pointing out that, when \( k \) increases, the average RMT decreases; the reason is that after the first CHI it is possible to exploit the beneficial effects of class caching and local class loading of the agent platforms employed.

6. Related work

Several approaches have been proposed in the literature for enabling interoperability among heterogeneous MASs. The taxonomy, shown in Fig. 15, classifies such approaches into three further-specializable main classes: specification-based, low-level and high-level. Each class is described in the following subsections, with a more detailed discussion of the high-level approaches to which the JIMAF approach proposed in this paper belongs.

6.1. Specification-based approaches

Specification-based approaches provide interoperability ruled by standard specifications such as MASIF (Mobile Agent Systems for Interoperability Facility) [25] and/or FIPA (Foundation for Intelligent and Physical Agents) [11].

The MASIF specification contains a detailed conceptual model that was agreed upon by five MAS providers (Crytaliz/MuBot, IBM/Aglets, the Open Group/MOA, GMD Focus/JMAF-Magna, and General Magic/Odyssey) under the umbrella of OMG. Nevertheless, its focus is on the interoperability of MASs through the adoption of two main interfaces: MAF-AgentSystem, for agent transfer and management; MAF-Finder, for naming and location. Furthermore, as designed under the umbrella of OMG, MASIF is tightly coupled to CORBA. It is worth noting that the MASs developed by the MASIF promoters were not made interoperable through MASIF.

The FIPA specifications focus more on the standardization of agent systems than on mobile agent systems even though the design of an interface between FIPA-based and MASIF-based agent systems has been defined.

A CORBA-based approach that combines the FIPA communication standard with MASIF was proposed in [3].

Despite such standardization efforts, most of the currently available MASs are compliant to neither MASIF nor FIPA, so must be heavily modified, both in their architecture and programming model so to comply with such standards. Therefore, although the benefit of such an approach involves compliance with standards, the necessity to modify too many already existing MASs which are not compliant with such standards makes this approach hardly impracticable for providing MAS interoperability (see Table 7).

6.2. Low-level approaches

Low-level approaches support interoperability by adding application-level adaptation layers to and modifying the existing software components of the MASs to be made interoperable.
The concept of the universal server, able to host a mobile agent coming from any MAS, and the use of software converters between MASs, which are capable of converting a mobile agent coming from a source MAS to a mobile agent of a target MAS, are research efforts belonging to this class. In particular, the actualization of the universal server concept has been presented in [29] where the agent server of the SeMoA mobile agent system is able to host mobile agents coming from JADE [4], Tracy [6] and Aglets [1,21]. Although this proposal provides useful design and architectural patterns, their application to other MASs require time-consuming customization.

The use of converters has been reported in [32] in which the authors present a way of converting an Aglets agent into a Voyager agent (and vice versa), so that the agent can migrate between the Aglets and Voyager platforms. Generally, the use of converters implies the implementation of a converter for each pair of MASs.

Although these approaches are based on voluntary interoperability, they require low-level modifications (i.e. at source code level) of the employed MAS which can alter backward compatibility and thus require MASs source code to be open, which is unlikely in commercial systems (see Table 7).

6.3. High-level approaches

In contrast to specification-based and low-level approaches, high-level approaches offer interoperability by means of application-level adaptation layers which are added to the MAS without modifying the MAS and without requiring access to the MAS source code. The definition of a universal agent, able to run on any MAS, and the creation of a generic interface (also known as the middleware approach) are research efforts belonging to this class. The former has been exploited in [26], whereas the latter in [16,23,15,7,5].

In particular, [26] uses the metaphor of a “head” that contains the agent logic or abstracted mission, and a “body” that serves to translate migration, communication, and lookup calls for the specific platform in question. The agent head would need a different body for each different MAS. This approach only allows for the migration of the “MONADS” agents to different agent platforms (currently Grasshopper, JADE, and Voyager), as the head is a MONADS agent.

In [16] a high-level approach is briefly documented which provides an “Abstraction Layer” able to allowing for interoperability of migration and execution among Aglets, Grasshopper, and Voyager.

In [23] the Guest proposal is presented which requires the mobile agent to be written so to use a Guest interface which is layered on top of the native interface of each mobile-agent platform. Existing mobile agents must be rewritten to use the Guest interface. The Guest approach only delivers migration and execution interoperability between the following MASs: Aglets, Grasshopper, and Voyager.

In [15] GMAS (Grid Mobile-Agent System) is proposed, which, in its simplest version, requires existing mobile agents to be rewritten to use a standard interface. GMAS, however, builds on this simpler functionality and allows existing agents to run unchanged atop different mobile-agent platforms. Each agent uses the native interface of its “home” platform, and “adaptors” (Foreign2GMAS

![Fig. 15. Taxonomy of the available approaches for MAS interoperability.](image)

<table>
<thead>
<tr>
<th>Classes</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification-based</td>
<td>Compliance to standards</td>
<td>Too many already existing MASs which are non-compliant with such standards</td>
</tr>
<tr>
<td>Low-level</td>
<td>Reuse of existing MAS and voluntary interoperability</td>
<td>Source code required. Modifications can alter backward compatibility</td>
</tr>
<tr>
<td>High-level</td>
<td>Reuse of existing MAS with no modifications of the MAS themselves; high flexibility</td>
<td>The addition of software layers atop already existing MAS could impair efficiency</td>
</tr>
</tbody>
</table>

Table 7 General comparison of the available approaches for MAS interoperability.
and GMAS2Native) translate the agent from one native interface to another when the agent is visiting a foreign platform. The adaptors first translate one native interface into a common interface, and then from the common interface to the target interface. Such dual translation prevents the combinatorial explosion in the number of adaptors. GMAS avoids the need for source code access, and uses only the public interfaces of each mobile-agent system. In particular, the Gateway component, used by foreign agents to move to other GMAS-capable MASs, serves as a conduit to the agent transport mechanism (discovery, marshaling and movement) at the source, while the Launcher component handles incoming launch requests, unmarshaling, and execution support for newly arrived agents. GMAS provides migration and execution interoperability whereas communication interoperability is deemed to be of minor importance. In particular, GMAS currently grants interoperability between D’Agents [13], Nomads [31] and EMAA [22]. Indeed, the actual application of GMAS required one change to the existing implementations of each used native platform: a fallback mechanism was introduced to try the Gateway if the native transfer protocol failed. For this reason, the GMAS approach is classified between the low-level and the high-level approaches.

In [7,5] the authors propose a mobility service, named Kalong, which was designed to be independent of any existing agent toolkit and only requires that mobile agent classes implement the Java Serializable interface. Kalong was integrated into Tracy [5] and, currently, also into JADE [7]. In particular, the integration of Kalong as a new mobility service into JADE was straightforward without requiring any changes to Kalong or JADE. In addition, the migration performance of JADE agents can be significantly improved using Kalong. This innovative approach is, however, only able to provide migration and limited execution interoperability among MASs and has currently been applied to grant interoperability only between Tracy and JADE.

JIMAF belongs to the class of high-level approaches and, in particular, uses both the concept of universal agent and the middleware approach (see Fig. 15). The universal agent represented by the HLP of the IMA is adapted through the AL interface to the LLP of a given MAS (see Sections 3 and 4.1). This allows for execution interoperability at low-cost as the process of adapting HLP onto a new MAS is straightforward; in fact, the development of a new LLP is required – that is, a mobile agent class which extends the specific mobile agent class of the new MAS and implements the IAgent interface (see Section 4.1). The middleware approach is used for the communication and migration interoperability. RMI middleware (see Sections 3.2 and 4.1.2) is used to support communication interoperability whereas the migration is based on a bridging infrastructure capable of receiving an IMA from a source MAS and injecting it into a different target MAS (see Sections 3.3 and 4.1.4). The bridging infrastructure concept is similar to the Gateway-based architecture proposed by GMAS but is implemented without any modifications of the MASs used. JIMAF is therefore effective to provide interoperability of execution, migration, and communication. Moreover, to enable communication between interoperable mobile agents and platform-dependent agents, wrapper agents are provided which act as mediators capable of interacting both with IMAs by using JIMAF-based events, and with platform-dependent agents by using platform-dependent communication mechanisms (see Section 4.1.3).

In conclusion, in light of the strengths and drawbacks of the currently available high-level approaches, the reuse of existing MASs without any modifications offers a great advantage. In fact, although efficiency may be impaired by the addition of software layers atop MASs (see Table 7), the JIMAF approach can provide high flexibility along with backward compatible solutions.

6.3.1. Comparison

We hereby compare the pros and cons of the JIMAF approach with respect to the aforementioned high-level approaches on the basis of the following criteria (see Table 8):

(i) Interoperability type: JIMAF and MONADS grant interoperability of execution, migration and communication whereas Kalong is only focused on migration interoperability and the others offer only execution and migration interoperability.

(ii) Ease-of-use: JIMAF and MONADS are very straightforward to make new MASs interoperable whereas GMAS requires some modifications of the MASs which, however, also call for the deep knowledge of such MASs at source code level. The integration of Kalong into new MASs requires the deep knowledge of both Kalong and the new MASs at the API level.

(iii) Availability: currently, only JIMAF, GMAS and Kalong are publicly available on the Web.

(iv) Performance: only JIMAF, GMAS and Kalong have documented performance overheads with respect to the MASs made interoperable. In particular, Kalong and GMAS only provided documentation of migration performance: the maximum performance overhead measured for GMAS is 6500% on D’Agents, the maximum performance overhead measured for JIMAF is 133% on Aglets, whereas, Kalong, being basically a mobility service, is able to boost the performance of slow mobility service layer (as demonstrated for JADE [7]). Moreover, the performance of JIMAF in message transmission and lookup can be improved when an IMA interact with another IMA having the same type of LLP. Such an IMA can actually use the specific platform-based send and lookup primitives provided by its LLP.
Table 8
Comparison of the available high-level approaches for MAS interoperability

<table>
<thead>
<tr>
<th>High-level approach</th>
<th>Comparison criteria</th>
<th>Interoperability type</th>
<th>Easy-to-use</th>
<th>Availability</th>
<th>Migration performance overhead%</th>
<th>Applications reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMAS</td>
<td></td>
<td>Migration/Execution</td>
<td>Low</td>
<td>Yes</td>
<td>6500</td>
<td>Yes</td>
</tr>
<tr>
<td>GUEST</td>
<td></td>
<td>Migration/Execution</td>
<td>Medium</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Abstraction layer</td>
<td></td>
<td>Migration/Execution</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>MONADS</td>
<td></td>
<td>Migration/Execution/Communication</td>
<td>High</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>Kalong</td>
<td></td>
<td>Migration</td>
<td>Medium</td>
<td>Yes</td>
<td>−40</td>
<td>N/A</td>
</tr>
<tr>
<td>JIMAF</td>
<td></td>
<td>Migration/Execution/Communication</td>
<td>High</td>
<td>Yes</td>
<td>133</td>
<td>Partial</td>
</tr>
</tbody>
</table>

(v) Application reuse: As GMAS, JIMAF allows for the use/reuse of platform-dependent code and functionalities. In particular, JIMAF offers reuse through wrapping techniques which allow for the communication between interoperable mobile agents and platform-dependent agents (see Sections 4.1.3 and 4.2). These features are not provided by any other high-level approaches.

7. Conclusions

As widely agreed, the exploitation and diffusion of mobile agents would undoubtedly increase if mobile agent systems were interoperable. Although several standardization and research efforts have been devoted to enabling interoperability among MASs, the majority of the available MASs are, to date, not interoperable.

In this paper, we have presented a high-level approach for MAS interoperability which is enabled by JIMAF (Java-based Interoperable Mobile Agent Framework) which supports the interoperability of execution, migration and communication between heterogeneous Java-based mobile agent systems. The main advantages provided by JIMAF are the following:

- programmers can develop mobile agents independently of any particular MAS which will support them, thus enabling mobile agent-based computing across a range of computing domains which are based on heterogeneous MASs;
- new or other already existing MASs can be easily made interoperable by (i) further specializing the PMAL so to obtain new low-level parts of the interoperable mobile agents, and (ii) developing mobile agent bridges for such MASs;
- the reuse of platform-dependent code is easily provided through wrapper agents.

JIMAF was experimented to make interoperable Aglets, Ajanta, Grasshopper, and Voyager. The performance evaluation carried out with respect to the MASs made interoperable has shown that the expected overhead, which a high-level approach inevitably introduces, is acceptable in open, heterogeneous computing environments like those based on the Internet, where openness and interoperability are more important than high-performance. We have also compared our approach, on the basis of selected criteria, with other high-level approaches currently available in the literature and have shown that, overall, the JIMAF approach is more effective than the others for providing interoperability of execution, migration and communication.

The growing interest on MASs interoperability of industrial and academic communities is demonstrated by the new FIPA (Foundation for Intelligent and Physical Agents) initiative, which is being re-launched under the umbrella of IEEE, to promote a working group on mobile agents, to which the authors of this paper are contributing and whose main objective is to establish standard architectures and protocols for complete MASs interoperability of execution, migration, communication, and security. Ongoing research efforts are therefore devoted to: (i) making interoperable Tracy, Semoa and JADE; (ii) decentralizing the bridging infrastructure; (iii) designing a security infrastructure for interoperability.

References


