10-24-2002

Toward Dynamic Interoperability of Mobile Agent Systems

Arne Grimstrup
Dartmouth College

Robert Gray
Dartmouth College

David Kotz
Dartmouth College

Maggie Breedy
University of West Florida

Marco Carvalho
University of West Florida

See next page for additional authors

Toward Interoperability of Mobile-Agent Systems*

Arne Grimstrup[1], Robert Gray[1], David Kotz[1], Maggie Breedy[2], Marco Carvalho[2], Thomas Cowin[2], Daria Chacón[3], Joyce Barton[3], Chris Garrett[3], and Martin Hofmann[3]

[1] Dartmouth College
[2] Institute for Human & Machine Cognition, University of West Florida
[3] Lockheed Martin Advanced Technology Laboratories

Abstract. Growing recognition of the benefits of mobile agents in distributed systems, such as military C4ISR, has led to a proliferation of mobile agent systems. However, incompatibilities between proprietary systems prevent the greater potential benefits of ubiquitous mobile agent computing. In particular, agents cannot migrate to a host that runs a different mobile-agent system. Prior approaches to interoperability have tried to force agents to use a common API and so far none have succeeded. This goal led to our efforts to develop mechanisms that support runtime interoperability of mobile-agent systems. This paper describes the *Grid Mobile-Agent System*, which allows agents to migrate to different mobile-agent systems.

1 Introduction

Many mobile-agent systems have been developed, but with different proprietary Application Programming Interfaces (APIs) for the agents. This proliferation of incompatible APIs implies that agents developed for one agent platform cannot be run on, let alone migrate to, a system with a different agent platform. In our opinion, interoperability of platforms is essential for mobile agents to become a ubiquitous technology.

Prior approaches, such as MASIF [4], defined a standard API and required all platforms that wish to interoperate to then implement the common API. These approaches, however, have failed to encourage many systems to adopt the API. [1]

This paper describes initial results from ongoing work to enable interoperability of mobile-agent systems. We begin with a motivating application scenario. Section 2 then describes the overall design followed by implementation details in Section 3. Section 4 presents a cost-benefit analysis of the interoperability system. Section 5 discusses related work. In Section 6, we discuss some future directions for our approach. Finally, in Section 7, we conclude by presenting a summary of important lessons learned.

* This research was supported by the DARPA CoABS Program (contracts F30602-98-2-0107, F30602-98-C-0170 and F30602-98-C-0162 for Dartmouth, UWF, and Lockheed Martin respectively) and by the DoD MURI program (AFoSR contract F49620-97-1-03821 for both Dartmouth and Lockheed Martin). The contact author is Thomas Cowin `tcowin@ai.uwf.edu`.

[1] The Mobile Agent System List identifies systems that do and do not comply with MASIF and other standards. The list is at
`http://mole.informatik.uni-stuttgart.de/mal/mal.html`

N. Suri (Ed.): MA 2002, LNCS 2535, pp. 106–120, 2002.

1.1 Motivating Application

The need for interoperability between information systems is readily apparent in peace-keeping and disaster-response operations. In these situations, a coalition of civilian and military organizations, each with its own intelligence or other information assets, is formed on short notice and required to operate in areas where the fixed communication infrastructure has been severely damaged or completely destroyed. The field units are forced to rely on limited-power devices that use an unreliable low-bandwidth data link to communicate between themselves and to access information held within constituent organizations' headquarters.

Today, the advantages provided by mobile agents are well suited to the coalition's operational environment. By moving the computation to the information source or another server on the fixed network with stable power, battery power consumption can be reduced and bandwidth can be used more efficiently. These advantages are only available if every member of the coalition uses the same mobile-agent system, however.

It is highly unlikely that every military, governmental and non-governmental organization will agree to use the same mobile-agent system *a priori*. The need for rapid response also precludes the conversion of existing agents to a new platform or major changes to the software infrastructure of any member organization. Clearly, the most desirable solution would allow existing agents to operate on foreign mobile-agent system hosts without modification. Our goal is to provide an inter-operation standard that can be used to provide migration inter-operability between different mobile-agent platforms.

2 Overall Design

Our basic approach to interoperability is to allow foreign agents to execute in a non-native mobile-agent system (MAS) by translating the foreign agent's API into the local platform's API. Instead of creating pairwise translations of MAS APIs, we defined a single common interoperability API (IAPI) that supports agent registration, lookup, messaging, launching, and mobility. Then each group provided translators between their MAS API and the IAPI. In this manner, it was not necessary to rewrite each MAS to conform to a new API, but to write translators from each MAS to the IAPI, and conversely, from the IAPI to the MAS to support these specific agent operations (i.e., 2N adaptors vs. N system rewrites). [2]

Figure 1 shows the structure of the *Grid Mobile Agent System* (GMAS). In the diagram, we see a foreign agent and a native agent operating in three different MAS environments. Each layer presents an API to the layer above. A typical MAS implementation consists of three layers: the agent itself, the MAS API, and the network. Our design adds three new layers: foreign MAS API to the GMAS API translation (Foreign2GMAS), GMAS API to native API translation (GMAS2Native), and a common communication and discovery service. In addition, our design adds two additional components to each participating MAS, an Agent Launcher and a Gateway. A MAS that wishes its agents to operate on other MAS's must implement the Foreign2GMAS translator and the gateway service, while those who are willing to host foreign agents as well must implement the

[2] For readers familiar with the PBM image-translation tools, PBM uses the same approach.

GMAS2Native translator, the Agent Launcher, and the Gateway. The Gateway serves as the conduit to the transport mechanism at the source, while the Launcher serves the same function at the destination.

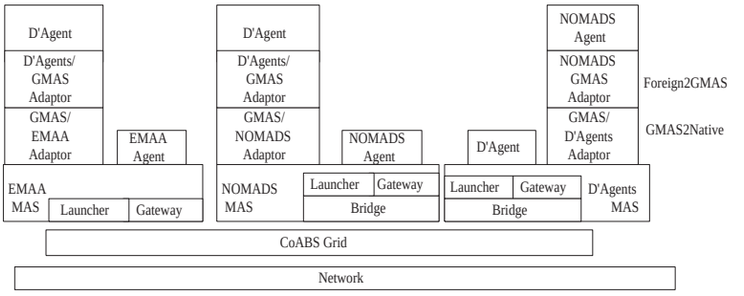

Fig. 1. Structure of GMAS

Since each MAS has its own communication and directory services, it is impractical to construct mappings between them. Instead, we rely upon a common substrate to provide communication and discovery services in this heterogeneous environment. In GMAS, we use the Jini-based DARPA Control of Agent Based Systems (CoABS) Grid [7], but CORBA or any other similar mechanism can also be used. NOMADS and D'Agents implemented their support of the Grid interface with the help of a bridge (a proxy that speaks both the Native and Grid protocols) due to lack of native support for Jini.

Foreign agents move to other GMAS-capable MAS hosts via their local Gateway and the Agent Launcher at the destination. The Gateway provides the discovery, marshalling, and movement operations using the local implementation of the GMAS API, described below. The Agent Launcher handles incoming launch requests, unmarshalling, and execution support for newly arrived agents. Depending on the network topology and local policy, an Agent Launcher may serve an entire subnet or an individual host machine and function as either a stand-alone service or as an integral component of the host mobile-agent system.

The local system composes the Foreign2GMAS translator with the GMAS2Native translator to map the Foreign API to the Native API. When a foreign mobile agent arrives at the system, the local system dynamically loads the corresponding Foreign2GMAS translator from a specified remote server. This capability is the key to providing interoperability of mobile-agent systems.

GMAS supports systems that are Java based, so any agent written in Java can operate upon a GMAS-enabled host utilizing standard Java API method calls, provided that its agency related calls (i.e., those of discovery, communication and migration) either conform to the GMAS API, or are translated to GMAS via a Foreign2GMAS adaptor. Access to certain Java methods may be circumscribed for host security reasons. An identifiable subset may evolve in common usage.

2.1 GMAS Interoperability API

The GMAS API provides methods to create an agent either by launching a new agent or by cloning the current agent on a remote host . The corresponding methods are *launchAgent()* or *cloneAgent()*. When launching a new agent, the agent's initial state must be provided to the system as a parameter. An explicit *moveAgent()* method is not provided, but can be easily implemented as a clone operation followed by the termination of the original agent.

Cloning an agent requires that the state of the agent be moved to the destination. One of the main advantages of using Java for mobile-agent programming is the ability to use object serialization for packaging an agent before shipment to another host. This behavior is supported in our design too. Any agent that implements Java's *Serializable* interface can be cloned through the GMAS API.

Many systems, such as D'Agents, do not support Java 2 serialization. To operate on those systems, an agent programmer must explicitly manage the conversion of the agent's data to and from an external form. These agents are referred to as *self-serializing* agents and must implement the *SelfSerializable* interface to be supported by GMAS.

The self-serialization process is straightforward. When a self-serializing agent attempts to move to a new host, the underlying mobility service retrieves from the agent a container holding objects that represent the agent's state using the *getAgentState()* method. Each object's value in the container is then converted to an XML-based representation containing its name, data type and value. An example of an agent's state after self-serialization is shown in Figure 2.

```
<gmas>
 <data request="11">
  <state>
   <var name="i" type="Integer" value="100" />
   <var name="f" type="Float" value="10.10" />
   <var name="b" type="Boolean" value="true" />
  </state>
 </data>
</gmas>
```

Fig. 2. SelfSerialized Agent State Transfer Message

To restore the agent state, this XML-based representation is parsed and a new container containing the state object is reconstructed from the triples. The *setAgentState()* method is invoked on the newly created agent instance with the container as the only argument and the values are restored to the agent's attributes.

To assist the programmer in the conversions, we provide the *AgentVariableState* class as a means of storing variables as well as handling the conversions to and from the message format. The agent programmer is still responsible for packing and unpacking the *AgentVariableState* with appropriate data objects.

The Agent Launcher handles cloning an existing agent and launching a new agent. In both cases, the Launcher creates a new instance of the agent on the receiving side. When cloning, the Launcher then copies the state of the original agent into the newly created agent. When launching a new agent, the launcher copies the initial state into the new agent.

The agent's meta information, including such items as its origin and the location of its class files, resides in an *AgentMetaData* object, and is provided to GMAS upon creation of the agent's GMAS representation. The system may obtain access to the meta data by invoking the *getMetaData()* method.

2.2 Gateway to Launcher Communication Protocol

As in many agent systems, agent migration depends on message passing. In our agent transfer, the client Gateway uses a two-phase protocol to launch or clone an agent. During the first phase, the Gateway sends the agent's meta information in the form of the *AgentMetaData* object to the remote Agent Launcher. If the remote Launcher decides to allow the migration, it returns a one-time-use token to the Gateway. The Gateway can then use the token to send the state of the agent, in either binary *(Serializable)* or XML *(SelfSerializable)* form, thereby completing the migration. This approach allows the receiver to perform authentication or other checks (such as server load) before granting the sender the right to send the agent.

3 Implementation

To evaluate our design, we implemented the inter-operable agent infrastructure for three different mobile-agent systems: D'Agents [6] from Dartmouth College, EMAA [3,9] from Lockheed-Martin Advanced Technology Laboratory, NOMADS [11] from the University of West Florida Institute for Human and Machine Cognition, and a reference implementation based on an unmodified Java 2 virtual machine. We first defined the communications protocol between the gateways and the launchers. Next, we implemented the major components: the launchers, the gateways, and the adaptors. Finally, we made a few minor changes to the agent systems so they would to work with the interoperability infrastructure. All of the above components are described in detail below.

Communication Protocol. As described in the design section, agent migration is accomplished via message passing between the source and the destination host. The CoABS Grid provides a well-integrated message-passing service, so implementing the launch request and response protocol only required that we create a Message object and invoke the appropriate message transmission method.

In our design, we specified that a launch request is comprised of a description of the agent and that agent's data state. Because we used the Grid to handle our communications, we were free to select between a string-based or a binary-based message format since a Grid Message is equally capable of carrying either form of data. We used a string-based approach since it would be easier to construct protocol bridges to systems that could not directly use the Grid software. Using a string-based format requires special marshaling

and unmarshalling code at both ends of the communications link. Rather than write custom software to handle the translation, we defined a set of XML tags that hold the agent description and state information.

Using an XML-based format gave us a structured way to translate agents to and from the launch request format, a way to leverage existing XML parsers such as Xerces[3], and a way to make changes in our data format with minimal disruption to the existing code base. XML does not support transfer of binary data, however, so we use Base64 to encode all serialized agents before inserting them into a launch request.

Launcher Implementation. Our launcher is implemented as a stationary Grid service that accepts and processes incoming Grid messages bearing launch requests. It can stand alone or run as a separate thread within a mobile-agent system's JVM.

When a message is received, the launch request is parsed and the agent information is extracted. The agent is returned as either a serialized object or a two-object set: one containing the agent metadata and the other containing the data state. The required Java class files are then downloaded from the specified source locations, and the agent is deserialized or the data state is loaded into a new instance of the agent.

The newly reconstructed agent object is now ready for execution. The launcher invokes an executor method, which is responsible for resuming the given agent object at the specified entry point. Depending on the executor selected, control may pass to the agent, a new thread may be created for the agent to run in, or the agent may be forwarded to another host. The behavior and types of executors available is dependent on the policy and configuration of the site where the launcher resides. Sites that have a single launcher supporting multiple MAS hosts will require an executor that forwards the incoming agent to the correct destination. Other sites may wish to present different execution environments to newly arrived agents based on an agent's native MAS. For example, an external launcher that supports a D'Agents MAS host may be configured to execute incoming EMAA and NOMADS agents as separate threads in its JVM due to possible code incompatibilities.

Gateway Implementation. The Gateway hides any local implementation details, such as special translation, from the agent programmer. In our reference implementation, this process is straightforward; the Gateway's *launchAgent()* and *cloneAgent()* methods find the destination then create and send the launch request to the appropriate launcher.

Since each platform provides a different Gateway implementation, the mobile agents should not carry a Gateway implementation object with them. Instead, the IAPI provides a wrapper class that returns the correct instance of the Gateway depending on the current location of the agent. The Gateway implementation returned is determined by a system property that can be set on the command line or in a configuration file. Using this additional layer of abstraction also allows the Launcher to change the returned mobility service implementation (i.e., Gateway) at runtime, allowing greater adaptability to changing local conditions.

Adaptor Implementation. The adaptors make up the last component of the implementation. As mentioned above, there are two kinds of adaptors: Foreign2GMAS and

[3] http://xml.apache.org/xerces-j/index.html

GMAS2Native. All three mobile-agent systems have implemented both adaptors. The remainder of this section describes the NOMADS implementation of the adaptors.

The NOMADS native API consists of a base *Agent* class that allows an agent to access all of the services provided by the underlying platform, including the mobility service. NOMADS supports both strong and weak mobility and hence provides two separate mobility services. Given that GMAS provides for only weak mobility, that is what we will address here.

When a NOMADS agent requests a move operation to a GMAS-enabled host, the mobility service encapsulates the agent into a *WrapperAgent* that conforms to the GMAS specification. Therefore, the *WrapperAgent* cloaks the native NOMADS agent in a GMAS shell, strictly for the purpose of getting successfully launched onto the destination host. This WrapperAgent's job is complete once the NOMADS agent is running on the destination host.

The Foreign2GMAS adaptor consists of an alternate implementation of the NOMADS base *Agent* class, providing access to the GMAS equivalent services while the agent is actually running, mapping any mobility requests made by the NOMADS agent into the GMAS API. The agent uses this alternate agent class as long as it executes on (and moves to) GMAS-enabled platforms. If the agent moves back to a NOMADS platform, the original base *Agent* class is utilized.

The GMAS2Native adaptor is utilized only when a GMAS Agent or a Foreign Agent composed with its Foreign2GMAS adapter is resident on a NOMADS system and makes a GMAS call, such as *cloneAgent()*. The local implementation of the Grid Mobility Service, in other words, the GMAS2Native adaptor, will be utilized to translate this into action within the NOMADS system, and clone the subject agent on its destination.

Modifications needed for Existing Systems. One of the key challenges in the design and implementation was to minimize changes to the existing mobile-agent systems. In NOMADS, we had to make only one change to the existing implementation. In our design, when an agent is running on its native platform, the agent uses the native platform's API (i.e., the *Agent* class) directly without going through any adaptors. When the agent decides to move to a GMAS-supported platform, we run into a problem because the native platform's API implementation does not know about the GMAS-supported platform.

Therefore, the one necessary change in each native platform implementation was a fallback mechanism to try the gateway if the native transfer protocol failed. Note that an alternative would have been to modify the agents to use the interoperability API as a fallback but our goal was to not make any changes to the code of the mobile agent.

3.1 Mobile-Agent Systems

We chose the particular MAS's we did because they were readily available to us and because they represent a range of design choices that impact interoperability. Since a complete presentation of each mobile-agent system is outside of the scope of this work, the relevant features of each system are summarized in Table 1.

Table 1. Relevant features of systems used in our experiments.

Feature	D'Agents	NOMADS	EMAA	GMAS Ref.Impl.
a) strong/weak	strong	strong and weak	weak	weak
b) JVM version	1.0.2	Aroma and 1.3.1	1.3.0-02	1.3.1
c) JVMs used	multiple	multiple	one	one
d) what moved	all code, data, stack	data, stack	data	data, code
e) code caching	no	yes	preinstalled	no
f) encoding	custom, fat	custom	serialized	serialized, custom
g) communication	sockets	sockets	sockets	Grid Messages
h) socket reuse	no	no	yes	no
i) security	off	off	off	none

(a) Both D'Agents and NOMADS implement strong mobility, which moves the code, data, *and control* states of an agent between host machines. Supporting strong mobility required modifications to the Java VM to extract the state information. Since weak mobility systems such as EMAA operate in unmodified JVMs, we could not expect state-capture support to be available in other systems. Therefore, GMAS was forced to use a weak mobility model.

(b) The modifications made to support the strongly mobile D'Agents were made to an older version of the JVM. The Grid is based on a more recent release of Java and uses new technologies such as Jini in its core services. NOMADS supports both strong and weak mobility APIs. The weak mobility version of NOMADS (NOMADS-Spring) is written in pure Java and executes on the standard Java VM. The strong mobility version of NOMADS (NOMADS-Oasis) uses a clean-room implementation of the JVM, called AROMA, that is mostly compatible with JDK 1.2.2 but does not support Jini. D'Agents and NOMADS-Oasis required communication bridges between the servers and the remainder of the Grid to interact with the Grid. As a result, both these systems pay a performance penalty when agents jump due to the translation and additional handling at the bridge.

(c) D'Agents and NOMADS-Oasis both create a new JVM process for each arriving agent. NOMADS-Spring, the GMAS reference implementation, and EMAA only create a new thread for each arriving agent, which is a much faster operation.

(d) The GMAS reference implementation moves agent code via the Java class loader. The bytecodes for an agent would be downloaded from the machine specified in the agent's metadata description. The downloaded code is not cached by the GMAS server. EMAA uses a similar mechanism that, when combined with caching, greatly reduces the transmission overhead. NOMADS also caches agent code. D'Agents has no caching support and must bring everything along as it jumps.

(f) D'Agents uses a custom encoding to carry agents from host to host but, due to the use of the older VM, cannot accept Java 2 serialized objects. NOMADS has the option to use either a custom encoding for its native agents or the regular Java serialization while

EMAA uses the object serialization services from Java exclusively. A GMAS-enabled agent may use either method described previously, but agents that use the *SelfSerializable* transfer mechanism suffer a greater performance penalty due to the additional overhead resulting from manual serialization.

(g) Since the Grid messaging system is based on RMI, GMAS suffers from higher per-message latency than the other socket-based systems.

(h) EMAA reuses sockets when an agent jumps, which saves connection setup and tear-down overhead. D'Agents and NOMADS create new sockets for each jump.

(i) The GMAS system has no support for encryption or other security features at present. Furthermore, the security features of D'Agents, EMAA and NOMADS are dramatically different, and it is unclear to what extent they could be made interoperable. In our experiments each system's security mechanism was disabled to ensure a fair comparison.

4 Performance Results

We measured three types of system-to-system jumps for each agent system. Each agent carried a common cargo of specifically identified data types, i.e., the same size payload. The first set of measurements were based on the native mobile-agent system running on each host.[4] We recorded the round-trip times as the native agent jumped back and forth, using the mobility provided by its mobile-agent system. Second, we measured self-serializable GMAS Reference Implementation agents and, last but not least, Java-serializable agents. In all cases, we measured the average round-trip time over 100 round trips. The results are summarized in Figure 3.

EMAA showed that its *Native* jump was considerably faster than a jump using GMAS (i.e., those times indicated by the *SelfSerializable* and *Serializable* bars in the EMAA section), by a factor of between 5 and 6, depending upon the type of GMAS mobility. NOMADS-Oasis showed even more distinction in this Native/GMAS jump time, by a factor of 10. This difference is attributable to some extent to the fact that the Aroma VM is restarted upon the agents' arrival at each system, and the agent's class must be retrieved via a URLClassLoader each time. This repeated retrieval is not necessary on systems that use Sun's VM, as the VM persists and the class remains cached. NOMADS-Oasis also showed a significantly slower overall time due to the unoptimized VM. Although not shown in the graph, we discovered that the CoABS Grid has significant startup overhead. Running an experiment with a freshly started Grid system added between 2 and 6 seconds to the first round-trip time.

The *launchAgent()* operation in the D'Agents environment is almost 100 times slower than native calls. Like NOMADS-Oasis, D'Agents cannot directly communicate via the

[4] Each host was a Gateway 9300XL laptop computer with a Pentium III 500MHz processor, 128 MB of RAM, 6 GB hard disk drive and a WaveLAN Gold 11 Mbps wireless network adapter. All hosts ran Slackware Linux version 7 with the 2.2.13 kernel, JDK v1.3rc1, Jini v1.1, and the CoABS Grid v2.0.0beta.

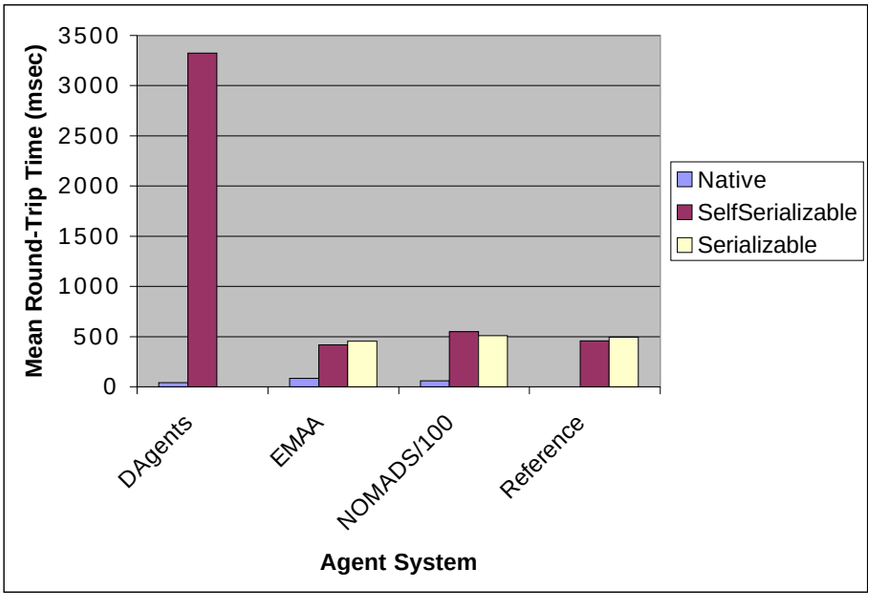

Fig. 3. Comparison of native vs. interoperable mobility operations. Note NOMADS transfer times were scaled down to fit on this graph.

Grid so we had to use a communication bridge to manage the translation. While this accounts for some of the performance penalty, the D'Agents implementation suffered in other areas. In the GMAS Reference Implementation, we used the URLClassLoader to handle the transfer of the agent's class file. This mechanism was not available in JDK 1.0.2 so D'Agents was forced to use a less efficient means. Also due to compatibility problems, we could not use our XML parser and were forced to use slower string manipulation methods to convert between GMAS messages and internal objects. Improvements in these areas should significantly cut the overhead cost for interoperable D'Agents.

We also measured the performance of the adaptors implemented for NOMADS and compared that with the native migration of agents in NOMADS. For this test, we used NOMADS-Spring and measured three different migration times: NOMADS-NOMADS (using the native migration protocol), NOMADS-GMAS (Reference Implementation), and GMAS-GMAS. In the NOMADS-GMAS test, an agent moves from a NOMADS-Spring platform to a GMAS platform and back. In the GMAS-GMAS case, the Foreign2GMAS adapter was needed on both hosts. Therefore, the agent is wrapped and unwrapped during the course of each iteration. Note that the same Native NOMADS agent was used in each of these different tests. Figure 4 shows the results of the experiments.

So, it is apparent that there is a cost, not insignificant in some cases, for interoperability. If it takes your agent an extra 3-400 milliseconds to move between platforms, of which at least one is a foreign mobile-agent system, and provided some service or benefit to your agent that cannot be obtained on your native systems, that seems a small

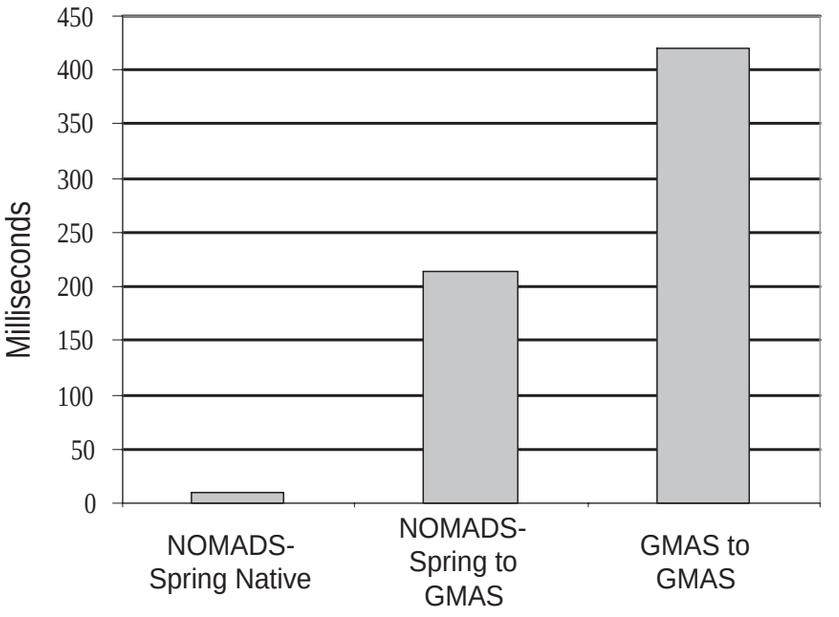

Fig. 4. Comparison of native vs. interoperable mobility operations in NOMADS-Spring.

price to pay for this increase in utility. Of course, after tuning these implementations, the performance will improve.

5 Related Work

Here we explain why we propose an interoperability standard, and how GMAS differs from the standards defined by the Object Management Group (OMG) and the Foundation for Intelligent Physical Agents (FIPA).

We recognize that there are some fundamental requirements for interoperability among heterogeneous mobile-agent systems. These include (1) discovery of agents and services, (2) communication, and (3) migration. Without all of these together, the benefits of interopability will not be realized.

The OMG Mobile Agent Facility (MAF) [12] builds on the CORBA naming, life cycle, externalization, and security services. It is intended to establish standards that support interoperability. This facility promises a degree of interoperability through common interfaces to two basic mobile-agent system components: the MAFAgentSystem and the MAFFinder. A MAFAgentSystem implementation provides agent management and transfer. The MAFFinder interface defines operations for registering and locating agents and agent systems.

The specification assumes, however, that it is rare that two different agent systems can receive and execute agents from each other. Indeed, the operation *get_nearby_-*

agent_system_of_profile() is provided to find only those migration targets that are running compatible agent systems. In contrast, our GMAS approach directly addresses the issue, providing a standard to enable participating heterogeneous mobile-agent systems to receive and launch each other's agents.

Like OMG MAF, GMAS leaves it up to the compliant agent system implementations to address security issues specific to mobile-agent systems and does not yet provide standard interfaces or implementations. Also like OMG MAF, GMAS does not specifically address transferring agents between agent systems written in different programming languages. This issue is true for MAF despite the fact that the underlying CORBA standard supports remote procedure calls among objects written in different programming languages. Mobility across heterogeneous programming languages is much more complicated, since mobile agents must execute their code on the heterogeneous platform directly.

GMAS defines a rich, declarative representation of the mobile agent in transit, to provide adequate flexibility in the mobility infrastructure for the heterogeneous MAS's. This contrasts with the minimal representation used by the MAF consisting of agent name, agent profile, agent (which can include class definition and state) , place name, class name, code base, and agent sender. GMAS adds agent meta data that are unnecessary in the more homogeneous environment assumed by OMG MAF, such as the name of the start method, and the agent's programming language. GMAS structures the agent in transit as an XML message to facilitate interpretation by heterogeneous implementations instead of a set of method call parameters. The additional metadata and the XML representation take GMAS a big step towards language independence. We have shown GMAS-enabled interoperability among three agent systems running on three different versions and implementations of the Java virtual machine.

FIPA defined a set of standards that represent a blueprint for constructing agent systems [5]. A few compliant agent system implementations are listed on the FIPA Web page,[5] but FIPA's standards have not been universally accepted.

The FIPA mobility specification recognizes two extreme cases of the mobility protocol. At one end of a spectrum, agents using *Simple Mobility Protocols* communicate a single `move` request to their local agent platform and the agent platform (system) takes care of moving the agent. At the other end of the spectrum, agents using the *Full Mobility Protocols* communicate with both the remote and the local agent platform and direct every stage of the move from remote request to local termination. Our GMAS model allows a spectrum of migration protocols that fall between the Simple and the Full Mobility Protocols.

GMAS aspires neither to provide standards as broad as FIPA nor to comply with a particular standard at a time when no universally accepted standard has emerged. For example, when an agent wants to find a remote peer, FIPA specifies a directory service, MAF its MAFFinder, and GMAS uses the CoABS Grid agent look-up service that is based on Sun's Jini. GMAS does not regulate communications protocols among agents but it assumes that heterogeneous agents are able to communicate with one another. Although we utilized the communication services provided by the Grid to craft GMAS, CoABS Grid adoption is not an absolute requirement: it would be equivalent to replace

[5] http://www.fipa.org/resources/livesystems.html

the directory and communication services with CORBA. The point is that you have to have some common substrate from which to work.

The mobile-agent description specified by FIPA is closer to our GMAS representation than the OMG MAF. It contains a parameter specifically designed to hold the agent's state. It does not address the needs of the two types of itinerant agent representations for both Self-Serializable and Serializable mobility as defined above.

Three recent papers describe mechanisms that allow mobile agents to migrate between distinct mobile-agent platforms. Magnin et al. [8] require the mobile agent to be written to use a *Guest* interface, which is layered on top of the native interface of each mobile-agent platform. In this way, it is quite similar to the simplest version of GMAS, where existing mobile agents must be rewritten to use a standard interface. GMAS, however, builds on this simpler functionality and allows existing agents to run unchanged on top of different mobile-agent platforms. Each agent uses the native interface of its "home" platform, and "adaptors" (Foreign2GMAS and GMAS2Native) translate 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 a combinatorial explosion in the number of adaptors. Tjung et al. [13] present a way of converting an Aglets agent into a Voyager agent (and vice versa), so that the agent can migrate between Aglets and Voyager platforms. The conversion requires access to the platform source code, however, something that is unlikely for commercial systems. GMAS avoids the need for source code access, and uses only the public interfaces of each mobile-agent system. Misikangas and Raatikainen [10] have a similar approach in that they do not require that each MAS be rewritten to a new interoperability API. They use the metaphor of a "head" that contains the agents logic or abstracted mission, and a "body" that serves to translate migration, communication, and lookup calls to the specific platform in question - instead of GMAS's GMAS2Native adaptor. The agent head would need a different body for each different MAS. This approach provides only for the migration of their "MONADS" agents to different agent platforms, however, as the head is a MONADS agent.

Although GMAS is not unique in its ability to enable agent migration among heterogeneous host agent systems, it allows mobile agents to use the native interface of their home platform, and in particular, existing agents do not need to be rewritten. Moreover, GMAS is scalable in the number of mobile-agent systems, since only two adaptors are required per agent platform, rather than one adaptor per each different pair of platforms. GMAS efficiently addresses the issues of packing, transferring, and unpacking an agent in a platform independent manner, and translation between APIs of different mobile-agent systems.

6 Future Work

This section briefly discusses our anticipated future work in the areas of security, resource control, and agent management.

The current GMAS implementation completely ignores security. We expect that secure communication and secure agent transmission is provided by the communication

transport layer. GMAS, for example, can take advantage of the secure messaging features of the CoABS Grid, while a CORBA-based implementation could exploit CORBA's security infrastructure. We do plan to incorporate basic agent authentication mechanisms into the Interoperability API. Mobile-agent systems have different security models however, so we expect that providing full security interoperability will be difficult.

While most mobile-agent systems recognize the need for resource control, once again different systems have different models and capabilities. For example, D'Agents supports a market-based approach to resource allocation and control whereas NOMADS supports a policy-based approach. NOMADS provides fine-grained control over resource usage (based on capabilities in the Aroma VM) whereas D'Agents and EMAA are constrained by the capabilities in the standard Java VMs. Therefore, coming up with interoperability mechanisms will be challenging. Our first step towards this goal is to make explicit the resource requirements of agents and to provide mechanisms that allow agents to query resource availability.

Agent management and system management are important in large-scale agent systems. Here again, current agent systems support different models and capabilities making it difficult to manage heterogeneous agent systems as a single administrative unit. In NOMADS, we are exploring the domain management capabilities of the KAoS agent architecture [1,2], which is scalable to multiple agent systems.

The next phase of development is to provide a common messaging API and enhance the adaptors to map the native messaging APIs of the three mobile-agent platforms to the common messaging API.

7 Conclusion and Lessons Learned

This paper describes our design for runtime interoperability of mobile-agent systems. The first stage, described here, involved defining an interoperability API that supported agent migration and agent messaging.

The current implementation operates over the DARPA CoABS Grid, which provides the basic registration, lookup, and messaging infrastructure. The performance for the three mobile-agent systems are slower by a factor of 10, but the system proves that interoperability is possible.

The distinguishing feature of our approach has been not to force a common API on all mobile-agent platforms. We recognize that past efforts using this approach have failed. Instead, our goal is to embrace the diversity of the different platforms and to create the necessary translation mechanisms to allow the systems to interoperate. While we have successfully demonstrated interoperability for agent messaging (via direct CoABS Grid adoption) and agent migration, we also recognize that several tricky issues remain to be solved.

The limitations to interoperability stem from the wide range of models and features of different mobile-agent systems. Mobile-agent systems are still at the stage where each system stresses different capabilities while ignoring others. One system difference for which we have no solution is strong versus weak mobility.

Another lesson involved incompatibilities in the Java API. For example, D'Agents currently uses JDK 1.0.2 whereas EMAA and NOMADS support Java 2. Therefore,

although we can move an agent from NOMADS or EMAA to D'Agents, the agent may fail to execute because of the differences at the level of the Java API.

Overall, we are pleased with the ability of GMAS to cleanly enable agent migration among diverse agent platforms. The slow performance will improve with careful optimization. Nonetheless, for many applications, performance may be less important than the ability to obtain new services and benefits available only by migration to a foreign MAS platform.

References

1. Jeffrey M. Bradshaw, Stewart Dutfield, Pete Benoit, and John D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In Jeff Bradshaw, editor, *Software Agents*, pages 375–418. AAAI/MIT Press, 1997.
2. Jeffrey M. Bradshaw, Niranjan Suri, Alberto K. Cañas, Robert Davis, Kenneth Ford, Robert Huffman, Renia Jeffers, and Thomas Reichherzer. Terraforming Cyberspace. *IEEE Computer*, 34(7):48–56, July 2001.
3. Daria Chacón, John McCormick, Susan McGrath, and Craig Stoneking. Rapid application development using agent itinerary patterns. Technical Report #01-01, Lockheed Martin Advanced Technology Laboratories, March 2000.
4. D. Milojicic et al. MASIF: The OMG mobile agent system interoperability facility. In *Proc. of the Second International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer-Verlag, September 1998.
5. FIPA Architecture Board. *Agent Management Support for Mobility Specification*. Foundation for Intelligent Physical Agents, Geneva, Switzerland, June 2000.
6. Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, January 1998.
7. Martha Kahn. *CoABS Grid User's Manual*, Available from http://coabs.globalinfotek.com/public/downloads/Grid/documents/GridUsersManual.v3.3.0.doc. Global InfoTeK Incorporated, October 2000. Version 2.0.0beta.
8. Laurent Magnin, Thang Viet Pham, Arnaud Dury, Nicolas Besson, and Arnaud Thiefaine. Our Guest agents are welcome to your agent platforms. In *Proceedings of the Seventeenth ACM Symposium on Applied Computing 2002 (SAC 2002)*, Madrid, Spain, March 2002.
9. Susan McGrath, Daria Chacón, and Ken Whitebread. Intelligent mobile agents in the military domain. In *Proceedings of the Autonomous Agents 2000 Workshop on Agents in Industry*, Barcelona, Spain, 2000.
10. Pauli Misikangas and Kimmo Raatikainen. Agent migration between incompatible agent platforms. In *Proceedings of the Twentieth International Conference on Distributed Computer Systems*. IEEE Computer Society Press, April 2000.
11. Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong mobility and fine-grained resource control in NOMADS. In *Proc. of ASA/MA2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 2–15, September 2000.
12. The Object Management Group. *Mobile Agent Facility*, January 2000. v1.0.
13. D. Tjung, M. Tsukamoto, and S. Nishio. A converter approach for mobile-agent system integration: A case of Aglet to Voyager. In *Proceedings of the First International Workshop on Mobile Agents for Telecommunication Applications (MATA '99)*, pages 179–195, Ottawa, Canada, October 1999.