Write Once, Move Anywhere: Toward Dynamic Interoperability of Mobile Agent Systems

Arne Grimstrup, Robert Gray, and David Kotz
Dartmouth College

Thomas Cowin, Greg Hill, and Niranjan Suri
University of West Florida

Daria Chacón and Martin Hofmann
Lockheed-Martin Advanced Technology Laboratory

Dartmouth College Computer Science
Technical Report TR2001-411

July 19, 2001

Abstract

Mobile agents are an increasingly popular paradigm, and in recent years there has been a proliferation of mobile-agent systems. These systems are, however, largely incompatible with each other. 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. Our goal, summarized in the catch phrase *Write Once, Move Anywhere*, led to our efforts to develop mechanisms that support dynamic runtime interoperability of mobile-agent systems. This paper describes the *Grid Mobile-Agent System*, which allows agents to migrate to different mobile-agent systems.

The contact author is Niranjan Suri <nsuri@ai.uwf.edu>. This research was supported by the DARPA CoABS Program (contracts F30602-98-2-0107 and F30602-98-C-0162 for Dartmouth and Lockheed Martin respectively) and by the DoD MURI program (AFoSR contract F49620-97-1-03821 for both Dartmouth and Lockheed Martin)

1 Introduction

There is increasing interest in the mobile agent paradigm. 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 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 [5] have attempted to define a standard API and require all platforms that wish to inter-operate to then implement the common API. However, these approaches have failed to encourage systems to adopt the API. [1]

This paper describes initial results from ongoing work to enable dynamic interoperability of mobile agent systems. We begin with a motivating application scenario. Section 2 describes the overall design followed by implementation details in section 3. Section 4 presents a cost-benefit analysis of the interoperability system. Section

[1] The Mobile Agent System List identifies systems that do and do not comply with MASIF and other standards.

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.

1.1 Motivating Application

Our motivating application arises in the context of an urban peacekeeping scenario, where the soldiers have been given the mission of finding and arresting a group of terrorists that are active within a particular city. As part of this mission, the analysts at mission headquarters must monitor intercepted phone calls for indications of suspicious activity. The application that does this monitoring makes use of two databases. The first database is a Black-Gray-White (BGW) database[2] that contains names, aliases, descriptions and affiliations of known terrorists, and the second database contains transcripts of intercepted phone calls along with associated annotations such as the time of the call, the source phone number, and the target phone number. The application first queries the BGW database to get the names and aliases of terrorists operating in that part of the world, and then examines each new intercepted phone call, scoring the call according to whether it was made at the right time, and whether the transcript includes any of the names or aliases or any suspicious words. Phone calls whose scores are above some threshold are reported to the analysts.

This application could be implemented in several ways, but two factors motivated applying mobile agents. First, the network connection between mission headquarters and the databases might be unreliable and of low bandwidth. The phone calls must be examined at or near the phone-call database, so that only high-scoring calls are sent across this unreliable, low-bandwidth link. The developers of the phone-call database might not have anticipated the needs of every client however, and thus the desired examination code might not be available at or near the database. Mobile agents can dynamically deploy the examination function to the phone-call database. Thus, the analyst's machine sends out a mobile agent to perform the monitoring task. The agent migrates to the BGW database and gets the description of known terrorists, and then proceeds to the phone-call database where it applies the examination function to each new phone call. The agent stays at the phone-call database for the duration of the mission, and sends all high-scoring calls back to the analyst for further review. Avoiding the transmission of irrelevant phone calls more than makes up for the transmission of the agent, and significant bandwidth savings are achieved over the mission lifetime. Moreover, the agent can continue analyzing phone calls even if the unreliable link back to mission headquarters goes down, and simply generate a queue of relevant phone calls, ready for transmission as soon as the network link comes back up.

This mobile-agent approach is straightforward if the databases and analyst machines belong to the same nation. If this nation's military has selected mobile agents as a valid implementation technique for military applications, presumably it has selected a particular mobile-agent system. Imagine, however, that the databases and the analyst machines belong to *different* nations, something that is common in peace-keeping missions, which are often multi-national coalition operations. It is unreasonable to assume that all participant nations would select the same mobile-agent system, but they might agree on an interoperation standard. The goal of our work was to develop a standard that allows mobile agents not only to communicate with agents from a different system, but also to migrate from one agent system to another. Allowing inter-system migration makes the mobile-agent phone application possible even when different mobile-agent systems are installed on the different machines.

2 Overall Design

Our basic approach to interoperability is to allow foreign agents to execute in a non-native mobile agent system by translating the foreign agent's API into the local platform's API. We did not wish to build $n \times n$ translators, how-

[2] A Black-Gray-White database contains descriptions of bad, neutral, and good people.

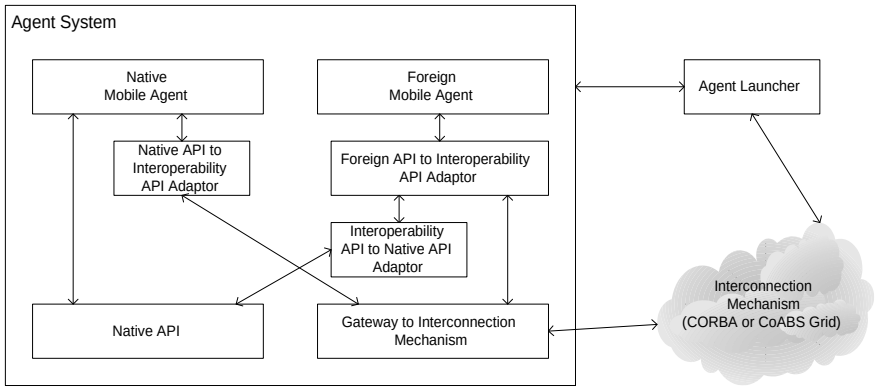


Figure 1: Structure of Mobile Agent Interoperability System

ever, so we followed the $(n-1) \times n$ approach. That is, we defined a single common interoperability API (IAPI) and then wrote translators to and from each of the local platform APIs and the IAPI. [3] The IAPI is defined through two different Java interfaces that support agent registration, lookup, messaging, launching, and mobility.

Figure 1 shows the overall design for the interoperability mechanism, referred to as the *Grid Mobile Agent Service* (GMAS) in this paper. We anticipate that some interconnection mechanisms will be present that can handle the network interconnection, registration of agents and systems, and the lookup of agents and systems. In our current implementation, the interconnection mechanism is the DARPA Control of Agent Based Systems (CoABS) Grid, which is Jini based, but we anticipate that CORBA or any other similar mechanism can also be used.

Each mobile agent system that wishes to interoperate must develop four components. These are: a Gateway to the Interconnection Mechanism (Gateway) that implements a common Interoperability API (IAPI), an adaptor that maps the native platform API to the IAPI (Native2IAPI), an adaptor that maps the IAPI to the native platform API (IPAI2Native), and a launcher service. Note that Figure 1 shows a fifth component, the Foreign API to IAPI adaptor (Foreign2IAPI). This is not provided by the local platform developer. Its role will be explained later.

The agent launcher is a key component of the system. The launcher registers the mobile agent system with the Grid and handles incoming requests from other agent platforms to launch or move agents within the local system. Depending on the network topology and local policy, a launcher may serve an entire subnet or an individual host machine as a stand-alone service or as an integral component of the host mobile agent system. Our design assumes the availability of a global name lookup service to provide agent and launcher discovery, and a reliable communication system between the various servers by which mobile agents can pass between hosts.

The second key component of the system is the Gateway, which provides an implementation of the IAPI on the local system. The IAPI is a set of interfaces and utility classes and are described below.

The last interesting component of the system is the Foreign2IAPI adaptor. This adaptor must have been made available to the system by the remote mobile agent system. The local system composes the Foreign2IAPI adaptor with the IAPI2Native adaptor in order 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 Foreign2IAPI adaptor. This capability is the key to providing dynamic interoperability of the mobile agent systems.

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

2.1 Interoperability API

Every inter-operable agent must provide a description of itself to the launcher on arrival. This description lists information about the agent including its origin and the location where the Java class files may be obtained. This information resides in a *AgentMetaData* object and must be provided at the time an agent is created. An agent (or the system) may obtain access to the meta data by calling the *getMetaData()* function.

The IAPI provides methods to create an agent either by launching a new agent or by cloning the current agent. The corresponding methods are *launchAgent()* or *cloneAgent()*. When launching a new agent, the agent's initial state must be provided to the system.

Note that IAPI does not provide an explicit *moveAgent()* method. The move can be 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 behaviour is supported in our design too. Any agent that implements Java's *Serializable* interface can be cloned through the IAPI.

Many systems, such as D'Agents, do not support Java serialization however. In order to operate on those systems, an agent programmer must explicitly manage the data store of the agent. These agents are referred to as *self-serializing* agents. To assist the programmer in that task, we provide the *AgentVariableState* class as a means of storing variables as well as handling the conversions to and from the message format. The IAPI defines the *SelfSerializable* in order to support agents and systems that cannot implement Java's serialization.

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 launch parameters into the new agent.

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 sends a launch request message to the destination server's Launcher, which responds with success or failure.

A launch request message has two forms. In the case of self-serializing agents, a request is comprised of two sections: a basic description of the agent that is being moved (the metadata) and the variable state of the agent. The metadata contains information such as the agent name and a unique identifier as well as operational information about the execution entry point and the location from which the agent code can be downloaded. The variable state section contains a listing of the name, type and value of all variables contained in the agent. In the case of Java serializable agents, the launch request carries a byte stream representation of the serialized agent object.

3 Implementation

To evaluate our design, we created a reference implementation of an inter-operable agent infrastructure and then added support for interoperability to three mobile agent systems: D'Agents, EMAA and NOMADS. In this section, we present our implementations and discuss the decisions made during that work.

3.1 Reference Implementation

Our GMAS reference implementation uses the Jini-based CoABS Grid [7] to provide the underlying lookup and communication services needed to support interoperability. On this foundation, we constructed the communication protocol, the launcher and the gateway components.

3.1.1 Communication Protocol.

As described in the design section, agent migration is accomplished via message passing be-

tween 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 us to 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. We did not specify a format for a launch request however. We chose to leave these decisions to the service implementers so they can choose a representation appropriate to their underlying communication mechanism. Because we used the Grid to handle our communications, we were free to choose between a string-based or a binary-based message format since a Grid Message is equally capable of either form of data. We chose 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 [10], 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.

3.1.2 Launcher Implementation.

Our launcher is implemented as a stationary Grid service that accepts and processes incoming Grid messages bearing launch requests. It can be run stand alone or 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 agent information is then passed to an inter-operable agent handler where the required Java class files are 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 agent object is then passed to an executor, which is responsible for resuming the agent at the specified entry point. Depending on the location of the server, the executor may pass control to the agent, fork a new thread for the agent to run in, or forward the agent on to another machine. Executor behaviour is dependent on the policy and configuration of the site where the launcher resides. An implementation may specify an executor for each agent system, or use a generic executor to support previously unknown agent types.

3.1.3 Gateway Implementation.

Each system that wishes to inter-operate using our mechanism must implement the Gateway. The gateway hides any local implementation details, such as special translation, from the agent programmer. In our reference implementation, this process is straightforward; the *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 allows the launcher to change the returned mobility service implementation at runtime allowing greater adaptability to changing local conditions.

3.2 D'Agents

D'Agents [6], formerly known as *AgentTcl*, is a mobile agent system that supports agents written in Tcl, Scheme or Java. Each site that accepts D'Agents must run a D'Agent server. The server is constructed in four layers. The base layer manages the network interactions of the server. The engine layer provides administration, migration, communication, and non-volatile storage required to support mobile agents. On top of the engine, the agent operations such as *jump* and *send* are implemented as a common set of agent APIs. Finally, a language interpreter, or virtual machine in the case of Java, connects an agent to the server via a set of stub routines that wrap the common agent APIs. The interpreters also protect the hosts from malicious agents and provide state capture and reconstruction facilities in addition to giving the agent the necessary runtime environment. Each agent executes its own interpreter in a separate process thereby isolating it from interference by other agents.

The version of Java supported by D'Agents was the main obstacle to implementing interoperable agents support on a D'Agents server. To capture the execution state of a Java agent, D'Agents required a specially modified Java virtual machine (JVM), based on the version 1.0.2 JVM. Our reference implementation, however, was implemented on top of the CoABS Grid, which uses Jini, for lookup and communication services. Since Jini requires language features and APIs that were added in later revisions of the Java language, our D'Agents server could not talk to any other agent system via this infrastructure.

We overcame this compatibility problem by constructing a simple communication bridge and using the implementation hiding abilities of our IAPI. Instead of making a connection to the Grid, the D'Agents implementation of the mobility service establishes a standard socket connection to the bridge and sends out its launch request. The bridge then takes that request, places it in a Grid message, and forwards that message to the destination host. The response message is then returned over the same path. The use of a string-based request format ensured that bridge construction was a straightforward process.

The language version incompatibility problem creates two limitations on interoperability. First, since object serialization is not supported in Java version 1.0.2, only non-serialized mobile agents can be accepted on or be launched from D'Agent hosts. This restriction can be handled at the launcher level by simply rejecting incoming serialized agents. The second restriction requires that incoming agents must be written to be compatible with Java version 1.0.2 since there is no support for newer language APIs and features. It may be possible to test and reject incompatible incoming agents, but that is beyond the current scope of our work.

3.3 EMAA

The Extendable Mobile Agent Architecture (EMAA) is a Java-based, object-oriented mobile agent architecture [4, 8]. At EMAA's core lies an agent Dock that resides on each execution host. The Dock provides an execution environment for agents, handles incoming and outgoing agent migration, and allows agents to obtain references to services. EMAA allows users to define agents, services, and events. EMAA agents are composed of small, easily reused tasks performed to meet a goal for a user. An agent's tasks are encapsulated in an itinerary; itineraries are structured as process-oriented state graphs. Agents may be mobile, and they typically make use of stationary services.

EMAA agents employ weak mobility; that is, the agent's full execution state (stack and program counter) is not transferred to the receiving machine; rather, the agent's data state is maintained, and the agent starts execution at a predefined entry point upon arrival. Because EMAA does not need to do state capture below the level of the Java Virtual Machine(JVM), it is able to use a standard, unmodified JVM and can evolve with JVM releases. An EMAA agent may migrate in between execution of activities in its itinerary. To migrate, an agent invokes the CommunicationServer, a core Server that is a part of the Dock, to serialize and send itself to another

machine.

Several types of EMAA agents exist, ranging from extremely simple to complex. The simplest EMAA agent merely implements the Agent interface, which extends the Serializable and Runnable interfaces. Such an agent is given its thread by the Dock and may use the Dock to send itself elsewhere or acquire references to service, but it does not have any structure or form imposed upon its execution. An intermediate type of EMAA agent, the SequentialAgent, has a sequential itinerary; that is, given a list of task/host pairs, it executes each task on its corresponding host, in order. The most complex, and most frequently used form of EMAA agent, the ComposableAgent, employs an activity diagram-based itinerary, and makes its own execution path and mobility decisions.

Because the EMAA agent system runs on a standard SUN JDK 1.2 virtual machine, we found it possible to use both the reference implementation of both the launcher and the client APIs. We envision that this should be possible for many other standard Java-based agent systems also. To incorporate the reference MobilityServer implementation into the EMAA agent system, we embedded it with an EMAA Server that was started by the EMAA Dock.

We tested the delivery of all three types of EMAA agents via the GMAS. However, we took two separate approaches: one for the simplest EMAA agents (which merely implement the Agent interface), another for the SequentialAgent and ComposableAgent. The latter two are classes that a user configures with tasks rather than fully coding on the spot.

In the case of the simplest agent, we constructed a class implementing both EMAA's Agent and GMAS' GridMobileAgent interfaces. As an implementor of the GridMobileAgentInterface, the agent class was able to generate metadata describing its run method, the machine to move to, etc. Using the GridMobileAgentSystem wrapper class, the agent obtained a class implementing MobilityServiceInterface and requested it to move the agent to another machine as described by an array of Entries. When received by a Launcher, the metadata was unpacked from XML, and the run method specified therein was called on the agent object.

In the cases of the SequentialAgent and ComposableAgent, we proceeded differently, because these agents are implemented as EMAA classes rather than interfaces. As such, they expect to use the EMAA CommunicationServer to move themselves. For these agents, it becomes obvious that either the agents themselves must be modified to become "GMAS-aware", or the EMAA CommunicationServer must be provided with back-end "shims" to a GMAS-compliant MobilityServiceInterface. To move the SequentialAgent using the GMAS, we encapsulated it within a class that implemented the GridMobileAgent and Serializable interfaces. This class in turn obtained a class implementing MobilityServiceInterface, and requested it to move the encapsulated SequentialAgent to the desired location. On the other end, the wrapper GridMobileAgent class called the run() method of the EMAA agent and ran it. We followed the same process for the ComposableAgent.

3.4 NOMADS

NOMADS [11] is a mobile agent system developed at the University of West Florida's Institute for Human and Machine Cognition (IHMC). Both strong security and mobility were fundamental design goals of this system. As Java was chosen as NOMADS language for its interoperability, and security and strong mobility were things that could not be affected by Sun's implementation of the Java VM, IHMC chose to implement their own Java VM, Aroma. NOMADS provides strong security for the host platform in that individual agents resource consumption can be monitored and limited or curtailed as required by the policy of the host.

The persistent NOMADS environment that actually executes on a given host is called an 'Oasis' and can host multiple agents concurrently, with each running its own (Aroma) VM. Strong mobility requires that the execution state of an agent (i.e., the thread state) be captured and moved from one host to another across a network so that the agent can take up execution right where it left off. NOMADS relies on Aroma's

state capture capabilities to provide strong mobility. As with D'Agents, NOMADS is limited to weak mobility when participating in the interoperable system.

As the Aroma VM does not support Jini, we did not implement the CoABS Grid API within NOMADS itself. As with D'Agents, we created a bridge agent to the Grid. This bridge runs on Sun's VM and allows NOMADS agents to use the Name Service and Communications features of the Grid. So, when a NOMADS Agent moves from system to system using the interoperability mechanism, the agent sends its message to the bridge first, which passes it through to the destination launcher via the Grid. In a NOMADS destination, the Launcher contacts the NOMADS Oasis environment, and passes the agent in for launching. As the agent is still an XML message, the Launcher submits an instance of a utility class that extracts the agent metadata and agent variable state information from the XML message. The Agent's class is loaded by the utility from the URL specified in the metadata, and the agent is then invoked at the entry method specified within the metadata. In the serialized case, the agent's class file is retrieved from the specified URL, the agent is deserialized or reconstituted, with its previous state intact, and then the agent's entry method is invoked.

4 Cost Benefit Analysis

4.1 System Development

Our goal in creating this inter-agent mobility is to design an elegant, scalable, and sufficiently flexible infrastructure that will allow agents from many systems to have, essentially, universal transport across all mobile agent systems. What we have done, on a smaller scale, is get D'Agents, NOMADS and EMAA working together with the underlying structure provided by the CoABS Grid, Jini and Java. The actual cost of designing and developing the GMAS infrastructure involved roughly six person-weeks of effort. The majority of the time was spent in developing the reference implementation, and a shorter time was required by each agent team to adapt the individual agent systems to the API. Each of our systems had pre-existing support and understanding of the CoABS Grid.

Now that the design and reference implementation are in place, it would be easy to adapt another mobile agent system to use GMAS and the CoABS Grid.

As agent systems evolve and mature, their efficacy will, to some extent, be predicated upon their ability to inter-operate with other existing mobile agent systems, and to accomplish the tasks for which they are designed as efficiently as possible. The effectiveness of the entire technology of mobile agents will be affected by the development of an interoperability standard, which will enable all of the adopting systems to communicate, locate diverse services, and quickly travel from platform to platform irrespective of underlying hardware, software, or Mobile Agent System.

Most mobile agent systems are research systems, and therefore do not endure all of the rigors of commercial software development. Consequently, it is difficult to quantify the cost/benefit tradeoffs of a proof of concept system such as GMAS. The potential long-term benefit of moving the mobile agent community closer to a common standard of interoperability is high, however, and should be a key goal for mobile agent system developers.

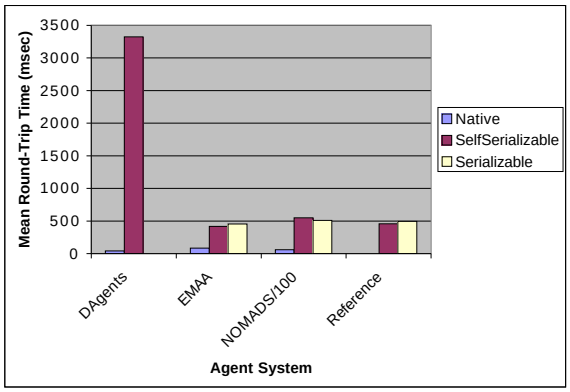
4.2 Agent Development

Once the APIs have been defined and melded into each mobile agent system, the incremental cost to the agent programmer is small. To use the non-serializable method of mobility requires some effort to explicitly pack up the agent's data prior to requesting the move, and to unpack and restore it upon arrival. If one can use Java serialization, the effort required is considerably less.

4.3 Agent Migration Times

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 na-

Figure 2: Comparison of Native vs. Interoperable Mobility Operations



tive mobile agent system running on each host[4]. The round-trip times were recorded as the native agent jumped back and forth, using the mobility provided by its mobile agent system. Second, SelfSerializable GMAS agents were measured and finally Java-serializable agents were measured. In all cases, we measure the average round-trip time over 100 round trips. The results are summarized in Figure 2.

Both the reference implementation and EMAA use Sun's Java VM; NOMADS uses an internally developed, unoptimized VM written to the Java Specification. EMAA showed that its native jump was considerably faster than a jump utilizing GMAS, by a factor of between 5 and 6, depending upon the type of GMAS Mobility. NOMADS showed even more distinction, with 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 retrieval is not necessary in systems that use Sun's VM. NOMADS also showed a significantly slower overall time due to the unoptimized VM. 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, D'Agents cannot directly communicate via the 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 implementation, we used the URLClassLoader to handle the transfer of the agent. This mechanism was not available in JDK 1.0.2 so we were 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 inter-operable D'Agents.

So, it is apparent that there is a cost, not insignificant in some cases, for interoperability. If it takes your agent an extra 3 or 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 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).

The OMG Mobile Agent Facility (MAF) [9] builds on the CORBA naming, life cycle, externalization, and security services. It is intended to establish standards that support interoperability among heterogeneous mobile agent systems. This facility promises a degree of interoperability through common interfaces to two basic mobile agent system components: the MAFAgentSystem and the MAFFinder. A MAFAgentSystem

[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.

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 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 provide standard interfaces or implementations yet. Like OMG MAF, GMAS does not specifically address transferring agents between agent systems written in different programming languages. This 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 the mobile agent in transit, because the receiving agent system need not be forced to make unnecessary assumptions about the incoming agent. This contrasts with the minimal representation used by the MAF consisting of agent_name, agent_profile, agent, 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 explicit agent state in the case of self-serializing mobility. 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 [1]. 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 through the CoABS Grid.

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 Type 1 and 2 mobility as defined above.

GMAS appears to be unique in its ability to enable agent migration among heterogeneous host agent systems. GMAS addresses precisely the issues of packing, transferring, and unpack-

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

ing an agent in a platform independent manner, and translation between APIs of different mobile agent systems.

6 Future Work

This section briefly discusses our future anticipated 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 interoperability in terms of security 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 [3, 2], which is scalable to multiple agent systems.

The next phase of development, underway, provides translation layers between native agent platform APIs and the interoperability API. The translation layers make it easier for a new mobile agent platform to become compatible with the set of existing platforms by providing just two translators: from the new native API to the interoperability API and vice-versa. We are currently implementing such translators for our three agent systems.

7 Conclusion and Lessons Learned

This paper describes our design for dynamic 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 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 learned was the 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.

References

[1] F. A. Board. *Agent Management Support for Mobility Specification.* Foundation for Intelligent Physical Agents, Geneva, Switzerland, June 2000.

[2] J. Bradshaw, N. Suri, M. Kahn, P. Sage, D. Weishar, and R. Jeffers. Terraforming Cyberspace: Toward a policy-based grid infrastructure for secure, scalable, and robust execution of Java-based multi-agent systems. In *Proc. of the Workshop on Agent-based Cluster and Grid Computing, at CCGrid 2001*, May 2001. Enlarged version to appear in IEEE Computer, in press.

[3] J. M. Bradshaw, S. Dutfield, P. Benoit, and J. D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In J. Bradshaw, editor, *Software Agents*, pages 375–418. AAAI/MIT Press, 1997.

[4] D. Chacón, J. McCormick, S. McGrath, and C. Stoneking. Rapid application development using agent itinerary patterns. Technical Report Technical Report #01-01, Lockheed Martin Advanced Technology Laboratories, March 2000.

[5] D. M. 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, Sept. 1998.

[6] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, January 1998.

[7] M. Kahn. *CoABS Grid User's Manual.* Global InfoTeK Incorporated, October 2000. Version 2.0.0beta.

[8] S. McGrath, D. Chacón, and K. Whitebread. Intelligent mobile agents in the military domain. In *Proceedings of the Autonomous Agents 2000 Workshop on Agents in Industry*, Barcelona, Spain, 2000.

[9] The Object Management Group. *Mobile Agent Facility*, January 2000. v1.0.

[10] T. X.-A. Project. Xerces java parser. http://xml.apache.org/xerces-j/index.html.

[11] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. 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.