

# An Overview of the NOMADS Mobile Agent System<sup>1</sup>

Niranjan Suri<sup>2</sup>, Jeffrey M. Bradshaw<sup>2,3</sup>, Maggie R. Breedy<sup>2</sup>, Paul T. Groth<sup>2</sup>, Gregory A. Hill<sup>2</sup>, Renia Jeffers<sup>3</sup>, Timothy S. Mitrovich<sup>2</sup>

## ABSTRACT

NOMADS is a mobile agent system that supports strong mobility (i.e., the ability to capture and transfer the full execution state of mobile agents) and safe Java agent execution (i.e., the ability to control resources consumed by agents, facilitating guarantees of quality of service while protecting against denial of service attacks). The NOMADS environment is composed of two parts: an agent execution environment called Oasis and a new Java-compatible Virtual Machine (VM) called Aroma. The combination of Oasis and the Aroma VM provides key enhancements over today's Java agent environments.

## Keywords

Mobile Agent System, Strong Mobility, Transparent Mobility, Java Virtual Machine, Resource Control, Security, Policy.

## 1. INTRODUCTION

Java is currently the most popular and arguably the most mobility-minded and security-conscious mainstream language for agent development. However, current versions fail to address many of the unique challenges posed by agent software. While few if any requirements for Java mobility, security, and resource management are entirely unique to agent software, typical approaches used in non-agent software are usually hard-coded and do not allow the degree of on-demand responsiveness, configurability, extensibility, and fine-grained control required by agent-based systems.

The security model in Java is rapidly evolving to provide some of the increased flexibility and fine-grained control required for agents. From the beginning, Java has featured a typed pointerless virtual machine instruction set, a bytecode verifier, class loaders, a security manager, and the concept of a "sandbox" to prevent executable code from accessing "dangerous" methods. Version 1.1 added an API for user security features such as signing of JAR archives. A major feature of the security model in the Java 2 release is that it is *permission-based*. Unlike the previous "all or nothing" approach, Java applets and applications can be given varying amounts of access to system resources based upon security policies created by the developer, system or network administrator, the end user, or even a Java program.

Despite these improvements, there is still much work that remains to be done. The ultimate goal is to define a set of standard underlying Java security policies and mechanisms that will make agent mobility as simple and safe as possible for both the agent and its host [12]. The mobile agent must be able to deal with situations where it has been shipped off to the wrong address, or to a place where needed resources are not available, or to what turns out to be a hostile environment [18]. Agent hosts may become unavailable or compromised at a moment's notice, and the agent may need to immediately migrate to a safe place or "die." Also, there is the very real possibility of unauthorized

inspection or tampering while the agent is traveling.<sup>4</sup> Agent hosts, on the other hand, have to deal with a variety of resource management issues.

The NOMADS agent system aims to support strong mobility and safe Java agent execution. The NOMADS environment is composed of two parts: an agent execution environment called Oasis and a new Java compatible Virtual Machine (VM) called the Aroma. The combination of Oasis and the Aroma VM provides two key enhancements over today's Java agent environments:

1. The ability to capture and transfer the full execution state of mobile agents (*strong mobility*). This allows agents to be moved "anytime" at the demand of the server or the agent rather than just at specific pre-determined points.
2. The ability to control the resources consumed by agents thereby facilitating guarantees of quality of service while protecting against denial of service attacks (*safe execution*). Adding these resource control capabilities to the access control mechanisms already provided by the new Java 2 security model allows mobile agents to be deployed with greater confidence in open environments.

In the following sections, we describe the requirements for strong mobility and safe agent execution in the context of past research on these topics. We discuss the specific limitations of previous efforts that we are attempting to address in our own work. We then describe the Aroma VM followed by a discussion on capturing and transferring the agent state and enforcing resource controls. Then, we briefly describe the Oasis agent execution environment. Finally, we outline issues and directions for future work.

## 2. NOMADS Requirements

Until recently, each mobile agent system has defined its own approach to agent mobility. Though new proposals such as FIPA's agent mobility standards (<http://www.fipa.org>) and OMG's Mobile Agent System Interoperability Facility [16] are a step forward, some of the required elements of strong and safe mobility cannot be implemented without foundational support in the Java language standard.

---

<sup>1</sup> This research is supported in part by DARPA's Control of Agent-Based Systems (CoABS) program (Contract F30602-98-C-0170), the NASA Aviation Extranet (Contract NCA2-2005), and the National Technology Alliance (NTA)

<sup>2</sup> Institute for Human & Machine Cognition, University of West Florida, {nsuri, jbradshaw, mbreedy, pgroth, ghill, tmitrovi}@ai.uwf.edu

<sup>3</sup> Intelligent Agent Technology, Phantom Works, The Boeing Company, {jeffrey.m.bradshaw, teresa.z.jeffers}@boeing.com

<sup>4</sup> The general problem of protecting mobile agents against malicious hosts is not discussed here, but see [25] for a discussion of one promising software-only approach.

Many of the early mobile agent systems recognized the need for strong and safe mobility. For example, the short-lived Telescript system provided some of the features we are implementing in NOMADS [27]. In particular, Telescript provided transparent agent migration and resource usage control. Resources used by agents were accounted in terms of “clicks,” a unit of charge that was then billed to the user or deducted from a user’s account. During the same timeframe, other groups also developed other agent systems. Systems such as Agent TCL, ARA, and others [6; 9; 22; 26] were primarily based on the Tcl [20] [21] language and interpreter although they ultimately provided support for a variety of languages. AgentTCL and Ara were also able to save the execution state of the agent to provide strong mobility. A few systems provided varying degrees of resource usage control.

Today, there are many commercial Java-based mobile agent systems currently available such as ObjectSpace Voyager [19], Concordia from Mitsubishi Electric ITA Horizon Laboratories [17], Jumping Beans from Ad Astra (<http://www.JumpingBeans.com>), and Aglets from IBM [14]. While all of these systems provide the ability to transport an agent from one server to another across a network connection, none of these transport mechanisms are completely transparent; none allow full capture of agent execution state. The security measures provided in these systems are also not mature enough to enforce the level of fine-grained security and resource control we desire. This is due in part to the lack of underlying support in Java for some of these mechanisms.

## 2.1 Strong Mobility

Strong mobility is vital for situations in which there are long-running or long-lived agents and, for reasons external to the agents, they need to suddenly move or be moved from one host to another. In principle, such a transparent mechanism would allow the agents to continue running without any loss of their ongoing computation and, depending on circumstances, the agents need not even be aware of the fact that they have been moved (e.g., in *forced mobility* situations). Such an approach will be useful in building distributed systems with complex load balancing requirements. The same mechanism could also be used to replicate agents without their explicit knowledge. This would allow the support system to replicate agents and execute them on different hosts for safety, redundancy, performance, or other reasons.

Strong mobility requires that the entire state of the running agent, including its execution stack, be saved prior to a move so that it can be restored once the agent has moved to its new location. The standard term describing this process is *checkpointing* [23]. Over the last few years, the more general concept of *orthogonal persistence* has also been developed by the research community [2]. The goal of orthogonal persistence research is to define language-independent principles and language-specific mechanisms by which persistence can be made available for all data, irrespective of type. Ideally, the approach would not require any special work by the programmer (e.g., implementation of serialization methods in Java, the use of transaction interfaces in conjunction with object databases), and there would be no distinction made between short-lived and long-lived data.

One of the powerful features of Java as a programming language is that its bytecode format, which is interpreted or compiled on-the-fly by the Java VM residing on the host platform, enables us

to save execution state in a machine-independent format. In principle, the design of Java allows the execution state to be restored on machines of differing architecture [24]. A similar but somewhat less general approach was originally implemented in General Magic’s Telescript language [27]. While it is possible to achieve some measure of transparent persistence by techniques such as having a special class loader insert read and write barriers into the source code before execution, such an approach poses many problems [13]. First, the transformed bytecodes could not be reused outside of a particular persistence framework, defeating the Java platform goal of code portability. Second, such an approach would not be applicable to the core classes, which cannot be loaded by this mechanism. Third, the code transformations would be exposed to debuggers, performance monitoring tools, the reflection system, and so forth, compromising the goal of complete transparency.

Modern mobile agent systems that use standard Java VMs typically transfer the instance data of objects from the source platform to the destination platform and restart execution of the agent on the remote platform. The task of writing mobile agents is more complicated when systems do not provide strong mobility. Figure 1(a) shows a basic agent written using Aglets that moves from one platform to another and displays a message. Figure 1(b) shows a significantly simpler agent written using the NOMADS system that accomplishes the same task. Note that the complexity is not particular to Aglets but to any mobile agent system without strong mobility.

NOMADS is not the only Java agent system providing strong mobility. Sumatra [1] and Ara [15] are other mobile agent systems that also provide strong mobility. However, both of these systems

```
public class Example extends Aglet {
    boolean _theRemote = false;
    public void onCreate (Object init) {
        addMobilityListener(
            new MobilityAdapter() {
                public void onArrival (MobilityEvent e) {
                    _theRemote = true;
                }
            }
        );
    }
    public void run() {
        if (!_theRemote) {
            System.out.println ("On Source");
            dispatch(destination);
        }
        else {
            System.out.println ("On Destination");
        }
    }
}
```

Figure 1(a): Example Aglets-based Mobile Agent

```
public class Example extends Agent
{
    public static void main (String[] args)
    {
        System.out.println ("On source");
        go (destination);
        System.out.println ("On destination");
    }
}
```

Figure 1(b): Example NOMADS-based mobile agent

have been implemented by modifying Sun's implementation of the Java VM. The disadvantage is that the system can only be redistributed to others who also have a source-code license to the Java platform. Also, as far as we are aware, both of these systems use version 1.0.2 of the Java VM.

Designing and implementing a new Java compatible VM has both advantages and disadvantages. The biggest disadvantages are the difficulty of achieving both compatibility and efficient performance. Implementers are bound to face a challenge in both implementing the complete Sun Java specification and maintaining complete compatibility as the specifications evolve. However we feel that there are also some advantages in implementing a VM from scratch while taking into account specific needs of mobile agent systems. In particular, NOMADS is able to provide strong mobility even in the presence of multiple concurrent threads and to dynamically control resource allocation on a fine-grained level. Another major advantage is the ability to freely distribute the VM and agent system.<sup>1</sup>

Another approach to being able to capture execution state is to use a pre-processor that rewrites or instruments the Java bytecode. An example of such an approach is [8]. The system requires a pre-processor that instruments the code by adding state-saving code where necessary. During execution time, the system utilizes the exception throwing mechanism in Java to capture and save the execution state of the process. Restoring the state is accomplished by wrapping blocks of code with conditional statements that skip over code that has already been evaluated. A limitation of the code instrumenting approach is that only the thread requesting migration may be migrated transparently. If an agent has multiple threads, then the system does require that all threads periodically call a function that "polls" whether any of the other threads has requested a checkpoint. Therefore, this requires that agents with multiple threads structure their code within certain requirements to allow for strong mobility, which is a disadvantage. The advantage of this approach is the ability to use a standard VM.

## 2.2 Safe Execution

Mechanisms for monitoring and controlling agent use of host resources are important for three reasons [18]. First, it is essential that access to critical host resources such as the hard disk be denied to unauthorized agents. Second, the use of resources to which access has been granted must be kept within reasonable bounds, making it easier to provide a specific quality-of-service for each agent. Denial-of-service conditions resulting from a poorly-programmed or malicious agent's overuse of critical resources are impossible to detect and interrupt without monitoring and control mechanisms for individual agents. Third, tracking of resource usage enables accounting and billing mechanisms that hosts may use to calculate charges for resident agents.

Resource protection mechanisms are available at several levels to software developers, including those provided by the networking environment, hardware, the operating system, and the features of a high-level language. In the case of agent technology, however, a persuasive case can be made for the advantages of an approach

based on language-based protection primitives [11]. While such an approach limits the developer to a restricted set of languages that can be supported, the increased precision in specification of rights, the relative efficiency of rights amplification, the ability to analyze programs statically and not just at runtime, and the portability of the language-based approach argue strongly in its favor.<sup>2</sup> An additional advantage in the context of our work with DARPA[5], NASA [4], and the NTA is that most of the language-based mechanisms we describe below can be incorporated transparently into any Java-based agent framework with little or no code modification required.

Given its status as the most popular and most rapidly evolving general-purpose safe language for Internet and agent applications, Java is the best first target for a language-based resource management approach for the agent community. However, there is still much to do to make it suitable for the industrial-strength agent applications of the future. Although the Java 2 security model is a step in the right direction, we anticipate that agent developers will require ever-greater levels of flexibility and host systems will need ever-greater protection against vulnerabilities that could be exploited by malicious agents. It is likely that some of these features will ultimately require changes to the Java architecture, such as the inclusion of an explicit Resource Manager to complement the current Class Loader and Security Manager [7]. For example, while new iterations of the Java security model will increasingly support configurable directory access by supplying the equivalent of access control lists to the Java Security Manager, there is no way to impose limits on *how much* disk storage or *how many* I/O operations or *how many* simultaneous print jobs may be performed by agents. Nor are there ways of controlling thread and process priorities, memory allocation, or even basic functions such as the number of windows that can be opened. A unique opportunity of our research is to explore techniques for dynamic negotiation of resource constraints between agents and the host. In NOMADS, we are taking a two-pronged approach: one prong relying on features provided by standard Java mechanisms and security policies, and the other relying on special features of our own VM and agent framework implementations. In the next sections we describe some of the details of our approach.

## 3. AROMA VIRTUAL MACHINE

The Aroma VM is a Java compatible VM designed and implemented with the specific requirements of strong mobility and safe execution. The primary goals for Aroma were to support:

1. Capturing the execution state of a single Java thread, thread group, or all threads (complete process) in the VM
2. Capturing the execution state at fine levels of granularity (ideally, between any two Java instructions)
3. Capturing the execution state as transparently to the Java code executing in the VM as possible
4. Cross-platform compatibility for the execution state information

---

<sup>1</sup> The new Community Source licensing policies from Sun Microsystems may allow third-party developers to modify and distribute Sun's implementation of the Java VM.

---

<sup>2</sup> Approaches for overcoming two possible disadvantages associated with language-based approaches (rights revocation and performance) are discussed in [10].

5. Flexibility in how much information is captured (in particular whether to include the definitions of Java classes)
6. Easy Portability to a variety of platforms (at least Win32 and various UNIX/Linux platforms)
7. Flexible usage in different contexts and inside different applications
8. Enforcement of fine-grained and dynamically changing limits of access to resources such as the CPU, memory, disk, network, and GUI

The current implementation of the Aroma VM falls short of some of the goals listed above. The limitations are that the VM can only capture the execution state of all threads and not the execution state of a subset of the threads present in the VM. Also, only the disk and network resource limits have been implemented. These limitations will be overcome in future versions of the Aroma VM.

The Aroma VM is implemented in C++ and consists of two parts: the VM library and a native code library. The VM library can be linked to other application programs. Currently, two programs use the VM library – *avm* (a simple wrapper program that is similar to the Java executable) and *oasis* (the agent execution environment). The VM library consists of approximately 40,000 lines of C++ code. The native code library is dynamically loaded by the VM library and implements the native methods in the Java API. Both the VM and the native code libraries have been ported to Win32, Solaris (on SPARC) and Linux (on x86) platforms. In general, the Aroma VM should be portable to any platform that supports ANSI C++, POSIX or Win32 threads, and POSIX style calls for file and socket I/O. We plan to port the Aroma VM to WinCE-based platforms as well.

Users use a standard Java compiler (such as the one provided with Sun's Java Development Kit) to compile Java source code to run on Aroma. The Aroma VM loads and executes files in the standard Java class file format.

### 3.1 Capturing Execution State

Aroma is capable of capturing the execution state of all threads running inside the VM. This state capture may be initiated by either a thread running inside the VM or by an external thread. The former is useful when the agent requests an operation that needs the execution state to be captured. The latter is useful when the system wants the execution state to be captured (for example, to implement forced mobility).

For several reasons, we chose to map each Java thread to a separate native operating system thread. The other alternatives were to develop our own threads package (which would be platform specific and difficult to port) or use an existing threads package (which may or may not be available on different platforms). Also, mapping to native threads allows the VM to take advantage of the presence of multiple CPUs. Therefore, if a VM has two Java threads running ( $JT_1$  and  $JT_2$ ), then there are two native threads ( $NT_1$  and  $NT_2$ ) that correspond to  $JT_1$  and  $JT_2$ . If the execution state of the VM is captured at this point and restored later (possibly on a new host), then two new native threads will be created ( $NT_3$  and  $NT_4$ ) to correspond to the two Java threads  $JT_1$  and  $JT_2$ .

However, mapping Java threads to native threads complicates the mechanism of capturing the execution state. This is because when one Java thread (or some external thread) requests a state capture,

the other concurrently running threads may be in many different states. For example, other Java threads could be blocked trying to enter a monitor, waiting on a condition variable, sleeping, suspended, or executing native code. We wanted as few restrictions as possible on when a thread's state may be captured so that we can support capturing execution state at fine levels of granularity. Therefore, the implementation of monitors was carefully designed to accommodate state capture. For example, if a Java thread is blocked trying to enter a monitor, then there is a corresponding native thread that is also blocked on some IPC primitive. If at this point the execution state is captured and restored later (possibly on a different system and of a different architecture), a new native thread must resume in the same blocked state that the original native thread was in when the state was captured. To support this capability, the monitors were designed in such a way that native threads blocked in monitors could be interrupted and stopped and new native threads could take their "place" in the monitor at a later point in time. As an example, consider a native thread  $NT_1$  on host  $H_1$  that represents a Java thread  $JT_1$ .  $NT_1$  could be blocked because  $JT_1$  was trying to enter a monitor. The VM will allow another thread to capture the execution state at such a time and when the state is restored later, a new native thread  $NT_2$  (on possibly a new host  $H_2$ ) will be created to represent  $JT_1$ . Furthermore,  $NT_2$  will continue to be blocked in the monitor in the same state as  $NT_1$ .

Another requirement is the need to support multiple platforms. In particular, to support capturing the execution state on one platform (such as Win32) and restoring the state on a different platform (such as Solaris SPARC). The Java bytecode format ensures that the definitions of the classes are platform independent so transferring the code is not an issue. For transferring the execution state, the Aroma VM assumes that the word size is always 32-bits and that the floating-point representations are the same. With these assumptions, the only other major issue is transferring state between little-endian and big-endian systems. The Aroma VM writes a parameter as part of the state information indicating whether the source platform was big- or little-endian. The destination platform is responsible for byte-swapping values in the execution state if necessary.

One limitation is that if any of the Java threads are executing native code (for example, by invoking a native method), then the VM will wait for the threads to finish their native code before initiating the state capture. This limitation is necessary because the VM does not have access to the native code execution stack.

### 3.2 Enforcing Resource Limits

The native code library is responsible for implementing the enforcement of resource limits. The current version is capable of enforcing disk and network limits. The limits may be grouped into three categories: rate limits, quantity limits, and space limits. Rate limits allow the read and write rates of any program to be limited. For example, the disk read rate could be limited to 100 KB/s. Similarly, the network write rate could be limited to 50 KB/s. The rate limits ensure that a program does not exceed the specified rate for any input and output operations. For example, if a network write rate of 50 KB/s was in effect and a thread tried to write at a higher rate, the thread would be slowed down until it does not exceed the write rate limit.

Quantity limits allow the total bytes read or written to be limited. For example, the disk write quantity could be limited to 3 MB.

Similarly, the network read quantity could be limited to 1 MB. If a program tried to read or write more data than allowed by the limit, the thread performing the operation would get an `IOException`.

The last category of limits is the space limit, which applies only to disk space. Again, if a program tries to use more space than allowed by the disk space limit, then the VM would throw an `IOException`. Note that the disk space limit is different from the disk write quantity limit. If a program has written 10 MB of data, it need not be the case that the program has used up 10 MB of disk space because the program could have written over the same file(s) or erased some of the files that it had written.

To enforce the quantity limits, the native code library maintains four counters for the number of bytes read and written to the network and the disk. For every read or write operation, the library checks whether performing the operation would allow the program to exceed a limit. If so, the library returns an exception to the program. Otherwise, the appropriate counter is incremented and the operation is allowed to proceed. To enforce the disk space limit, the library performs a similar computation except that seek operations and file deletions are taken into consideration. Again, if an operation would allow the program to exceed the disk space limit, the library returns an exception and does not complete the operation.

To enforce the rate limits, the library maintains four additional counters for the number of bytes read and written to the network and the disk and four time variables, which record the time when the first operation was performed. Before an operation is allowed, the library divides the number of bytes by the elapsed time to check if the program is above the rate limit. If so, the library puts the thread to sleep until such time that the program is within the rate limit. Then, the library computes how many bytes may be read or written by the program in a 100ms interval. If the operation requested by the program is less than what is allowed in a 100ms interval, the library simply completes the operation and returns (after updating the counter). Otherwise, the library divides the operation into sub-operations and performs them in each interval. After an operation is performed, the library sleeps until the interval finishes. For example, if a program requested a write of 10 KB and the write rate limit was 5 KB/s, then the number of bytes that the program is allowed to write in a 100ms interval is

512 bytes. Therefore, the library would loop 20 times, each time writing 512 bytes and then sleeping for the remainder of the 100ms interval. One final point to make is that if a rate limit is changed then the counter and the timer is reset. This reset is necessary to make sure that the rate limit is an instantaneous limit as opposed to an average limit.

We are currently evaluating two different implementation schemes for the CPU resource control. The first approach is to have each native thread (that represents a Java thread) check the percentage of CPU used by that thread (by querying the underlying operating system) and then sleep if necessary. The check would be performed periodically after executing a certain number of Java bytecode instructions. The second approach is to have a separate monitoring thread that checks the CPU utilization of all the Java threads and then pauses the Java threads as necessary. In this case, the monitoring thread will be assigned high-priority and will wake up periodically to perform its monitoring.

Although we have not started implementing the memory resource control, we expect that it will be fairly straightforward since the VM handles all object instantiations. The VM can maintain a counter variable that keeps track of memory usage and throw an exception (or take other appropriate action) when memory usage is exceeded.

## 4. OASIS EXECUTION ENVIRONMENT

Oasis is an agent execution environment that embeds the Aroma VM. It is divided into two independent programs: a front-end interaction and administration program and a back-end execution environment. Figure 2 shows the major components of Oasis. The Oasis Console program may be used to interact with agents running within the execution environment. The console program is also used to perform administrative tasks such as creating accounts, establishing resource limits, and so forth. The Oasis process is the execution environment for agents. Among other things, it contains instances of the Aroma VM for running agents, a Policy Manager, and a Dispatcher. Each agent executes in a separate instance of the Aroma VM. However, all the instances of the Aroma VM are inside the same Oasis process, which results in all of the VM code being shared.

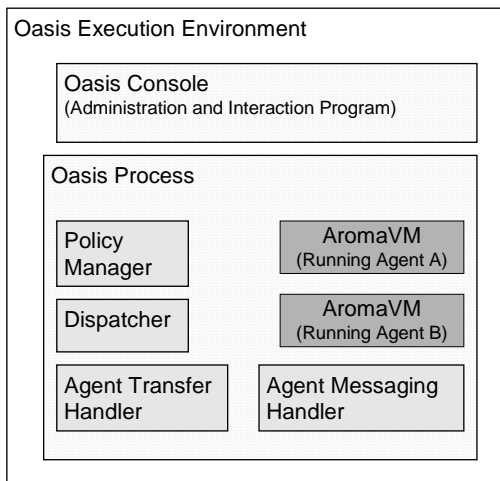


Figure 2: Oasis Agent Execution Environment

Category	Attributes
Agent Transfer	Maximum agent state size
	Maximum agent state transfer duration
Execution	Execution Duration
	Cloning Limit
Communication	Maximum incoming message queue size
	Maximum incoming message size
Access Control	Standard Java Security Manager Attributes
Resource Usage	Disk Data Transfer Rates / Sizes
	Network Data Transfer Rates / Sizes
	Memory Size
	Maximum Thread Count
	Maximum CPU Utilization (Percentage)

Table 1 : Parameters Controlled by the Policy Manager

Agents can communicate by sending and receiving messages. Message transfer is implemented by a simple API that provides a `sendMsg()` and a `receiveMsg()` functions. Agents are addressed via Universally Unique Identifiers (UUIDs). The messages themselves can contain any serializable Java object. Oasis maintains message queues for each agent. The message queues are maintained outside the Aroma VM instances so that it is easy to handle situations where a message arrives while a VM's state is being captured.

The Policy Manager is a major component of the execution environment. It is responsible for establishing security policies for all agents. Policies may be established on an individual account basis or on a group basis. The policy for each entity (individual or group) consists of four categories that address agent transfer, execution control, access control, and resource usage. Table 1 shows the specific attributes that may be specified for each of the four categories.

Upon startup, the Policy Manager reads a `policies` file. The `policies` file specifies resource limits that need to be applied to accounts. Figure 3 shows the format for the `policies` file.

As Figure 2 shows, a user or administrator may interact with the Oasis environment through a separate administration process that allows the user to examine and change various resource limits for an individual agent or a group of agents. The `policies` file specifies the policies for resource limits applied to an agent when the agent is received by Oasis. Each agent running within Oasis may have most of these parameters adjusted dynamically by the Policy Manager; in the case of intelligent agents, we anticipate that this may involve a process of negotiation. Some parameters such as `MaxAgentStateSize` and `MaxAgentTransferTime` are not changeable, since they do not apply once an agent is already executing.

Controlling so many parameters to set up a secure and foolproof environment requires a significant level of expertise on behalf of an administrator. We feel that in the near future, most administrators or maintainers of agent-systems are not likely to be

```
[Entry]
PolicyName=<polycyname>
MaxAgentStateSize=<bytes>
MaxAgentTransferTime=<milliseconds>
CloningLimit=<count; 0 => no cloning allowed>
MaxTimeToLive=<seconds>
MaxIncomingMessageQueueSize=<bytes>
MaxIncomingMessageSize=<bytes>
JavaSecurityPolicyFile=<file path>
DiskReadRateLimit=<bytes/millisecond>
DiskWriteRateLimit=<bytes/millisecond>
DiskReadQuantityLimit=<bytes>
DiskWriteQuantityLimit=<bytes>
DiskSpaceLimit=<bytes>
NetworkReadRateLimit=<bytes/millisecond>
NetworkWriteRateLimit=<bytes/millisecond>
NetworkReadQuantityLimit=<bytes>
NetworkWriteQuantityLimit=<bytes>
MemoryLimit=<bytes>
MaxThreads=<count; must be 1 or more>
```

Figure 3: Format for policies File

security experts but rather experts in some domain of application. To simplify the task of setting up secure environments, we are also working on an Agent Management Tool (AMT) that in addition to offering a convenient interface will also contain suggested security policies for typical scenarios [5]. Domain knowledge in the AMT can help agent designers determine what kinds of policies are appropriate for a given situation.

The AMT prototypes we have been building also provide a graphical interface for the monitoring, visualization, and dynamic control of resource usage at runtime so that certain agents in an application can have greater access to resources than others. The goal of the runtime interface is threefold: to guarantee some specified level of agent access or quality of service to agents providing critical functions; to minimize the possibility of unauthorized access or reduce the impact of denial-of-service attacks; and to provide the possibility of detailed resource accounting. In such a configuration, the AMT would communicate to the Policy Manager inside Oasis to gather information about executing agents and to control their limits.

The other component within Oasis worth mentioning is the dispatcher, which is responsible for executing an agent once its state information has been received. The dispatcher also enforces some of the execution policies such as the maximum length of time for an agent to live on the platform (`MaxTimeToLive`). In the future, the dispatcher will also support the notion of agents and agent groups executing within separate, isolated processes.

One important design choice was to run each agent within a separate instance of the Aroma VM. Such a design has both advantages and disadvantages. The advantage is that resource accounting and control is simplified. The disadvantage is increased overhead. We are working on the possibility of sharing class definitions between multiple Aroma VMs which should reduce the overhead significantly.

The policy manager is not responsible for enforcing any of the security policies. Instead, the actual enforcement is carried out by the agent transfer protocol handler for authentication and agent

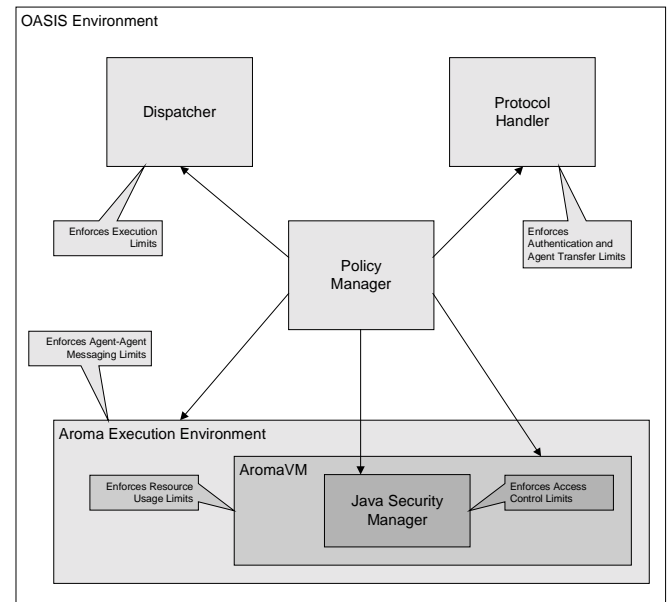


Figure 4: Security enforcement in Oasis

transfer control, the dispatcher for execution control, the Java security manager for access control, the Aroma execution environment for messaging control, and the Aroma VM for the resource usage control. Figure 4 shows the different components that enforce the various parts of the security policy established by the policy manager.

Security enforcement usually results in one of two consequences for agents. When an agent tries to perform an operation that would result in the agent exceeding a quantity limit (e.g., disk space, memory), an exception is thrown back to the agent using the standard Java exception mechanism. On the other hand, when an agent tries to perform an operation that would result in the agent exceeding a rate limit (e.g., disk usage rate, network transfer rate) the agent's request will be delayed until such time that processing the request would not result in the rate limit being exceeded. For example, if the agent's limit on network transfer rate is 10 KB/sec and the agent requests that a 100 KB block of data be transmitted, the Aroma VM will block the agent for 10 seconds while transmitting the data before allowing the agent to continue execution. Note that a rate limit is not based on averaging. A write rate limit of 10 KB/s does not imply that if the agent does not write anything for the first 10 seconds that it can write at 20 KB/s for the next 10 seconds.

One of the goals of Oasis is to allow dynamic adjustment of resource limits of agents. This causes a potential problem for certain kinds of resources when resource limits are lowered below the threshold already consumed by an agent. For example, an agent may have already used 10 MB of disk space and the resource limit might be reduced to 8 MB. The current implementation does not attempt to reclaim the 2 MB of disk space back from the agent. Instead, any future requests for disk space simply fail until the agent's disk usage drops below 8 MB. In the future, we would like to explore a mechanism to notify an agent about the change in resource limits (using a callback function) and allow the agent a fixed amount of time for the agent to comply with the changed limits or perhaps to negotiate some compromise. If the agent does not comply then Oasis has the option of terminating the agent or transmitting the agent back to its home or some other designated location.

## 5. CONCLUSIONS

We have described our motivations for developing a mobile agent system that provides both strong and safe mobility. We have also described the initial design and implementation of the Aroma VM and the Oasis agent execution environment. Initial performance results are promising. The speed of agent transfer using the current unoptimized NOMADS code ranges only from 1.27 to 1.71 times slower than the weak mobility competitors evaluated and we have several ideas for significantly increasing performance. The overhead for resource control in our experiment was less than 3%.

To date, both the Aroma VM and Oasis have been ported to Windows NT, Solaris (on SPARC), and Linux (on x86). The Aroma VM is currently JDK 1.2 compatible but with several limitations and missing features, the major omission being support for AWT. The AWT implementation affords opportunities for evaluating resource management mechanisms for graphical resources. Opportunities for further tests of NOMADS compatibility and performance will come over the next few months as we evaluate variations of a layered architecture incorporating different combinations of NOMADS elements in

conjunction with secure agent infrastructure collaborators at Boeing, Lawrence Berkeley National Laboratories (LBNL), and the National Institute of Standards and Technology (NIST) [12]. We will layer Boeing's KAOs framework [3] on top of NOMADS, providing support for high-level agent conversations, teamwork, and platform-independent domain and policy management using languages and tools aimed at users rather than developers [5]. We will continue to participate in efforts such as the FIPA architecture committee and the DARPA CoABS agent grid and mobility experiments [6] to ensure consistency with evolving standards and to make our work available to the community at large [5].

## 6. REFERENCES

- [1] Acharya, A., Ragnganathan, M., & Saltz, J. Sumatra: A language for resource-aware mobile programs. In J. Vitek & C. Tschudin (Ed.), *Mobile Object Systems*. Springer-Verlag.
- [2] Atkinson, M. P., & Morrison, R. (1995). Orthogonally persistent object systems. *VLDB Journal*, 4(3), 319-401.
- [3] Bradshaw, J. M., Dutfield, S., Benoit, P., & Woolley, J. D. (1997). KAOs: Toward an industrial-strength generic agent architecture. In J. M. Bradshaw (Ed.), *Software Agents*. (pp. 375-418). Cambridge, MA: AAAI Press/The MIT Press.
- [4] Bradshaw, J. M., Gawdiak, Y., Cañas, A., Carpenter, R., Chen, J., Cranfill, R., Gibson, J., Hubbard, K., Jeffers, R., Kerstetter, M., Mathé, N., Poblete, L., Robinson, T., Sun, A., Suri, N., Wolfe, S., & Bichindaritz, I. (1999). Extranet applications of software agents. *ACM Interactions*, in press.
- [5] Bradshaw, J. M., Greaves, M., Holmback, H., Jansen, W., Karygiannis, T., Silverman, B., Suri, N., & Wong, A. (1999). Agents for the masses: Is it possible to make development of sophisticated agents simple enough to be practical? *IEEE Intelligent Systems*(March-April), 53-63.
- [6] Cybenko, G., Gray, R., Kotz, D. & Rus, D. (2000). Mobile agents: Motivations, state-of-the-art systems, and frontiers. In J.M. Bradshaw (Ed.), *Handbook of Agent Technology*. AAAI/MIT Press, in press.
- [7] Czajkowski, G., & von Eicken, T. (1998). JRes: A resource accounting interface for Java. *Proceedings of the 1998 ACM OOPSLA Conference*, . Vancouver, B.C., Canada.
- [8] Fünfroeken, S. (1998). Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In K. Rothermel & F. Hohl (Ed.), *Mobile Agents: Proceedings of the Second International Workshop (MA 98)*, Stuttgart, Germany. (pp. 26-37). Berlin, Germany: Springer-Verlag.
- [9] Gray, R. S. (1996). Agent Tcl: A flexible and secure mobile-agent system. *Proceedings of the 1996 Tcl/Tk Workshop*, (pp. 9-23).
- [10] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., & von Eicken, T. (1998). Implementing multiple protection domains in Java. *Proceedings of the 1998 USENIX Annual Technical Conference*, . New Orleans, LA.
- [11] Hawblitzel, C., & von Eicken, T. (1998). *A case for language-based protection*. Technical Report 98-1670. Department of Computer Science, Cornell University, March.

- [12] Jansen, W. & Karygiannis, T. (2000). Mobile agent security. In J.M. Bradshaw (Ed.), *Handbook of Agent Technology*. AAAI/MIT Press, in press.
- [13] Jordan, M., & Atkinson, M. (1998). *Orthogonal persistence for Java—A mid-term report*. Sun Microsystems Laboratories.
- [14] Lange, D. B., & Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets.*, Reading, MA: Addison-Wesley.
- [15] Maurer, J. (1997) *Porting the Java runtime system to the Ara platform for mobile agents*. Diploma Thesis, University of Kaiserslautern.
- [16] Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., & White, J. (1998). MASIF: The OMG Mobile Agent System Interoperability facility. In K. Rothermel & F. Hohl (Ed.), *Mobile Agents: Proceedings of the Second International Workshop (MA 98)*, Stuttgart, Germany. (pp. 50-67). Springer-Verlag.
- [17] Mitsubishi (1999). Concordia <http://www.meitca.com/HSL/Projects/Concordia/whatsnew.htm>.
- [18] Neuenhofen, K. A., & Thompson, M. (1998). Contemplations on a secure marketplace for mobile Java agents. K. P. Sycara & M. Wooldridge (Ed.), *Proceedings of Autonomous Agents 98*, . Minneapolis, MN, , New York: ACM Press.
- [19] ObjectSpace (1999). ObjectSpace Voyager <http://www.objectspace.com/products/voyager/index.html>.
- [20] Ousterhout, J. K. (1990). Tcl: An embeddable command language. *USENIX Conference Proceedings*, Winter 1990.
- [21] Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. (Fourth ed.), Reading, MA: Addison-Wesley.
- [22] Peine, H., & Stolpmann, T. (1997). The architecture of the Ara platform for mobile agents. K. Rothermel & R. Popescu-Zeletin (Ed.), *Proceedings of the First International Workshop on Mobile Agents (MA 97)*. Springer-Verlag.
- [23] Plank, J. S. (1997). *An overview of checkpointing in uniprocessor and distributed systems focusing on implementation and performance*. Technical report UT-CS-97-372. Department of Computer Science, University of Tennessee, July.
- [24] Puening, M., & Plank, J. S. (1997). Checkpointing Java. In Online reference at <http://www.cs.utk.edu/~plank>:
- [25] Sander, T., & Tschudin, C. F. (1998). Protecting mobile agents against malicious hosts. In G. Vigna (Ed.), *Mobile Agent Security*. LNCS.
- [26] Suri, N., Ford, K. M., & Cañas, A. J. (1998). An architecture for smart internet agents. *Proceedings of the 1998 Florida Artificial Intelligence Research Society Conf (FLAIRS 98)*.
- [27] White, J. (1997). Mobile agents. In J. M. Bradshaw (Ed.), *Software Agents*. (pp. 437-472). Cambridge, MA: The AAAI Press/The MIT Press.