MAGIKMAIDS: Mobile Agents for In-Kernel Monitoring, Assessment and Intervention in Distributed Systems

Cristian Dinu  
University "Politehnica" of Bucharest  
Email: cristian.dinu@cti.pub.ro

Laura-Cristina Gheorghe  
University "Politehnica" of Bucharest  
Email: laura.gheorghe@cs.pub.ro

Nicolae Tapus  
University "Politehnica" of Bucharest  
Email: nicolae.tapus@cs.pub.ro

Abstract—As distributed systems have begun to play a more and more central role in modern computing, their size and complexity has increased up to the point where centralized, human-mediated management of the system is unfeasible. A need arises to endow these systems with a form of distributed intelligence so as to allow scalable self-organization and timely response to changing conditions. Mobile agents are one well-established method of implementing such versatile distributed intelligence and many mobile agent platforms have been created for use in computer networks and distributed systems.

While all such efforts so far have been limited to user space, we consider the novel approach of implementing such a platform in kernel space. We begin by making a case for why a kernel space implementation would present worthwhile advantages, then we investigate possible implementation paradigms and follow with the architecture and details specific to our chosen approach. Finally, we evaluate our platform in terms of performance and ease of use and draw the appropriate conclusions.

I. INTRODUCTION

Current trends in computing, as predicted by those such as Nessett[1], indicate that distributed systems are an increasingly pervasive paradigm in computing and our everyday lives. The Internet, massively multiplayer online games (MMPORPGs), wireless sensor networks and ambient intelligence are but a few of the numerous applications of distributed systems, and yet more will likely emerge as the boundaries of this technology are pushed by further research.

The increased pervasiveness of distributed systems necessarily leads to greater increases in their geographic scale and granularity, and thus to their complexity. Modern-day distributed systems may well feature node counts in the thousands or even hundreds of thousands[1], and it is soon apparent that manual, centralized, management of such systems is all but impossible. Solutions are needed by which this task can be delegated to intelligent automation within the system, likely distributed so as to still scale with such large node counts.

Mobile agents are one of the technologies by which distributed systems may be endowed with embedded intelligence. An agent, by definition, contains all the fledgling ingredients of intelligence, and a society of well-designed agents may feature, through emergence, more advanced behavior still, enabling timely and efficient reactions in a distributed system in response to an attack, a malfunction or a change in the nature of the computing task assigned to the system. Mobile agent societies scale well because they are inherently parallel, and represent a more efficient use of resources because they are able to carry functionality where it is needed, instead of it being built into every machine. In recognition of their potential, mobile agents in distributed systems have been put to use in tasks such as monitoring[2] [3], intrusion detection[4], checkpointing[5], and many others.

Although there have been many implementations of mobile agent platforms so far, to the best of our knowledge none has considered the possibility of running in the operating system kernel as opposed to being another user space application. Though running such a complex subsystem in kernel space poses considerable challenges in terms of security, safety and performance, we believe that such a scenario would present several unique advantages that make such an undertaking worthwhile.

First, the kernel environment offers unparalleled levels of control over the most intimate aspects of the operating system, a fact useful to agents that go beyond monitoring and reach into the realm of intervention. One can imagine an agent being able to tweak the algorithm of the system scheduler so that a particular load scenario can be most effectively addressed. Additionally, some monitoring tasks may require very precise timing or hardware operations that are the privilege of kernel space code. Kernel code that responds to a timer does not have to worry about being unpredictably delayed due to the scheduler electing a competing user space application.

Agents in the kernel are well protected against malicious user space interference. Indeed, the agent platform can be implemented such that a malicious user space application, even one with root privileges, cannot even detect the presence of the agents, let alone interfere with them. The fact that they have priority over any user space application also allows agents to police processes and prevent more indirect attacks such as fork bombs, resource hogging etc.

Finally, as mobile agents can implement drivers or even system services in a microkernel, they can form the basis of an exceptionally flexible upgrade system. Updating a driver on-the-fly may simply be a question of launching a mobile agent implementing the improved version and letting it diffuse throughout the distributed system on its own, automatically replacing the agent that implemented the previous version of
the driver.

This paper documents our efforts at implementing such a mobile agent platform - which we have named MAGIKMAIDS - in kernel space, more specifically the Linux kernel space. Before discussing our implementation, we first present some related work, in section II. Section III describes the main decisions and assumptions that shaped the ultimate design of the agent platform, while section IV goes into detail regarding the implementation of the platform at all levels: general architecture, API, programming details, etc. Section V presents a series of results regarding the performance characteristics and behavior of the MAGIKMAIDS platform under various test scenarios; and finally, in section VI we draw our conclusions as to the success of the experiment and any potential future work.

II. RELATED WORK

Our survey of the available literature shows that while there has definitely been much research into mobile agent platforms and applications in distributed systems (of which the aforementioned [2] [3] [4] [5] are but a small part), authors have steered clear of suggesting a kernel space implementation, likely because of the daunting challenges inherent to such a proposition, as well as the fact that it is only recently that technological advances in CPU speed and architecture have made it feasible to implement such a complex application in kernel space.

Nevertheless, there are some projects that touch upon at least some of the aspects important for a mobile agent platform in the kernel. Necula and Lee’s "Safe Kernel Extensions Without Run-Time Checking" [6], for instance, deals with a sort of non-mobile "agents" specifically designed to work in the networking subsystem of the kernel (as packet filters), and addresses the important problem of ensuring the safety and security of such code. The problem of mobile agent security, which is of the utmost importance for a platform situated in such a sensitive location as kernel space, is further discussed at length in Jansen and Karygiannis’ "Mobile Agent Security" [7], which we found to be an invaluable resource during the design phase for our platform in all matters that had security implications.

If a JVM-based agent platform implementation is considered, one work of particular importance in the matter is Okumura and Childers’ "Running a Java VM Inside an Operating System Kernel" [8], which presents their experience with implementing a functional Java Virtual Machine in kernel space, as well as an associated Just-In-Time (JIT) compiler. Their performance results prove that it is entirely feasible to run such a virtual machine in the kernel with only minimal performance degradation with respect to native code, provided that some reasonable restrictions are imposed upon the programs run in such a machine. We have also found a similar open-source project called TeaseMe, but it appears to have been abandoned for a number of years and the code likely requires considerable review.

III. DESIGN CONSIDERATIONS

Before we could start work on the details of the platform, a significant decision had to be made as to the major approach used for executing custom code, such as agents, in the kernel environment. There are two possible such approaches, each with its unique set of advantages and disadvantages. We will discuss them in the following sections and justify our final selection for the implementation presented in this paper.

A. Approach I: Agents as native code

This first approach involves treating agents as special Loadable Kernel Modules (LKMs) that, in addition to the standard kernel functionality, are also provided with access to agent platform functions that assist in environment discovery, migration, agent-to-agent cooperation, etc. Agents are coded in C, like any other LKM, and work somewhat like cpufreq or cpuidle governors: when the agent is first created - or arrives on a machine through migration - its module is loaded and the agent registers its callbacks while in its module initialization function so that it can receive agent platform-specific events. Upon migration or termination, the agent module is unloaded, not before giving the agent an opportunity to save its data to a binary stream so that it can be available for state restoration when the agent arrives on another machine.

This solution has several obvious advantages. First, agent performance is essentially identical to that of the rest of the kernel, thanks to the use of C code and the opportunity for compile-time optimizations. Memory and CPU utilization are optimal provided that the programmer is capable of producing reasonably efficient code. Second, agent code has a direct interface to all kernel functions available to a module. This aids in CPU efficiency, but also has implications for the agent platform complexity: the latter need only provide code to assist in migration, agent module loading/unloading, calling agent callbacks in response to events, and supporting agent communication.

Unfortunately, many more disadvantages spring to mind. Given that agents may have to travel across nodes featuring different kernel versions or configurations, a mechanism is needed in order to re-compile or re-link an agent as it arrives on a new machine. An agent could travel in the form of source code (obviously encrypted and signed) and be recompiled upon arrival, but even the compilation of a simple module is extremely resource-intensive, and the compiler toolchain would likely have to be extensively modified so that it can be invoked by the kernel without risking userspace interference. Alternatively, compilation could be offloaded to dedicated 'master' nodes in the distributed system, but this complicates the architecture and introduces numerous security issues (what happens if a 'master' is compromised?). Relinking has the potential to be much faster than recompilation, but it cannot handle differences in CPU architecture (such as the agent passing between machines of different endianness).

Moreover, because agent code executes natively, the agent platform is unable to prevent any bugs or malicious behavior on part of the agent from bringing down the entire system.
Agents have to be trusted to be friendly and nearly bug-free. For reasons such as the above, agent development would be very difficult, as the programmer would have to ensure safety in a language and environment notorious for their lack of such guarantees.

B. Approach II: Agents as Java bytecode

An alternative approach involves representing agents as Java bytecode that is executed in the context of a Java Virtual Machine (JVM) implemented in the kernel. Agents are written in Java (or any other language that produces compatible bytecode) and can access kernel and agent platform functions through special built-in classes (e.g. Magik, Kernel), while also providing methods for handling the events sent by the agent platform.

The disadvantages of this solution come in contrast to the advantages of the native method. Clearly, performance is reduced, possibly by several orders of magnitude, due to the necessity of interpreting the bytecode, as well as the additional abstraction layers. However, this may be mitigated by Just-In-Time (JIT) compilation, as well as by employing specialized JVM instructions that have begun to appear in modern CPUs (e.g. the Jazelle mode for ARM-based processors). Second, a complex adaptation layer between the Java environment and the kernel functions needs to be created, generally as part of the agent platform. This introduces overhead and generally limits the agent to only that functionality which has been explicitly included in this adaptation layer.

On the other hand, the virtualization layer resolves many of the problems that plagued Approach A. Agent portability is no longer a problem as Java bytecode easily transcends differences in kernel versions or machine architecture. Because agent code now runs in a virtual machine and all calls to the kernel pass through an adaptation layer, it is possible - and advisable - to perform sanity and security checks at every opportunity. The potential of a system-wide crash caused by a malfunctioning or malicious agent is greatly reduced. Finally, the Java language makes it easier to produce safe code, and complex programs can be implemented faster thanks to the object-oriented approach of the language as well as the many useful built-in classes.

Having weighed the advantages and disadvantages for both of these approaches, we decided that the most promising method was B), owing to the fact that a less than optimally efficient agent platform is still preferable to an insecure and dangerously unsafe one, and that recent technology offers several solutions for satisfactorily mitigating the performance impact characteristic to a JVM-based solution.

C. Founding principles for agents

To further help with limiting the performance impact of agents, as well as ensuring that potentially fatal programming flaws are avoided, we also set a number of guiding principles for designing MAGIKMAIDS agents:

- Agents will perform only brief, non-blocking operations in their callbacks. Any long-running or complex operations will be delegated to a tasklet or a separate thread, for which the MAGIKMAIDS platform should provide adequate functionality. This paradigm should be rather familiar to kernel developers (this is the classic way of writing a driver that responds to interrupts) as well as Android application programmers.
- Barring the callbacks, which will never be delayed, agent performance is expendable: their execution will be forestalled or slowed down if this is necessary for achieving minimal impact on userspace performance.
- Agents are completely isolated from each other; an agent cannot obtain a reference to any other agent running on the same machine, nor can he directly manipulate other agents through his interface to the agent platform. Agents will have to interact through a communication layer that agent instances may subscribe to voluntarily - thus, agents can only influence each other if they are specifically designed to do so, and any such communication can be adequately checked at the receiver before any action is performed.
- Agents shall minimize the allocation of objects on the heap using the new operator, particularly within repetitive sections. Whenever possible, preallocated objects will be reused. This reduces the strain on the memory allocation subsystem in the Linux kernel and minimizes the need for invocation of the garbage collector. Android application programmers should be quite familiar with this restriction.

IV. Implementation

A. Overall architecture

The overall architecture for a MAGIKMAIDS agent environment spanning an entire distributed system is that of perfect distribution: every node contains a MAGIKMAIDS agent platform installation (in the form of a LKM), and all discovery, migration, etc. operations are performed in a distributed manner. All nodes have the same rank - there are no 'master' nodes or other attempts at partial centralization. Agents may be injected anywhere in the network, provided that the user has the appropriate security privileges.

Akin to a network of routers self-discovering its own topology via neighbor discovery protocols and routing protocols such as OSPF, a collection of MAGIKMAIDS nodes can maintain system-level cohesion by having each node send packets to discover agent-enabled neighbors and exchange first-hand or propagated information with them. Communication between nodes is achieved via TCP connections managed by the Linux kernel networking subsystem, which has been further modified so as to enable special behavior for any traffic incoming on the MAGIKMAIDS ports. All communication is encrypted for confidentiality following a Diffie-Hellman exchange between every pair of agent-enabled nodes.

B. Per-machine architecture

Going into further detail, the architecture of a MAGIKMAIDS instance on any particular node is outlined in figure 1.
The main components shown are:

- **The agent platform LKM**, which sits on top of the Linux kernel and also includes any hooks or modifications done to the standard kernel infrastructure (e.g. in the networking subsystem). The LKM is responsible for directing all functions of the platform, such as agent environment maintenance, inter-node discovery and information exchange, implementing agent migration and communication, etc. The MAGIKMAIDS module also includes a JVM capable of executing Java bytecode and maintaining a heap where Java objects can be allocated and freed.

- **The JVM contexts** are isolated entities containing the set of classes that implement each type of agent (each type has its own JVM context). JVM contexts are created as each type of agent first arrives on the machine, and destroyed as a timer expires after the last agent of a given type has left the machine (thus, JVM contexts remain cached for a while in case an agent of that type returns). In addition to the classes specific to the agent, JVM contexts also contain a read-only mapping to the built-in class library that contains native implementations for system and utility classes such as `java.lang.String`, `java.util.ArrayList`, etc., as well as special classes that act as interfaces to the kernel or the MAGIKMAIDS agent platform.

- **The agent instances** contain state data for each individual agent instance. All agent instances of a given type are hosted in the corresponding JVM context. Because each agent instance is isolated and does not have direct access to any global variables, the heap as viewed from the context of any particular agent is private to that agent and part of the agent instance data structure. This leads to greatly increased efficiency in garbage collection and fault handling (e.g. when an agent terminates, all of its particular allocations can be immediately found and freed).

## C. Agent-platform-kernel interaction

Formally, interaction between an agent, the MAGIKMAIDS agent platform and the kernel takes places via user-programmed or automated calls made to a set of Java methods exposed by each entity. A diagram of this interaction is shown in figure 2.

![Agent, kernel and platform interfaces](image)

As depicted, the `Kernel` and `Magik` abstract classes represent an interface to the kernel and agent platform, respectively, as seen from the agent. An agent that wishes to terminate itself, for instance, would achieve this by calling the static function `Magik.terminate()`, which is ultimately mapped to an event in the agent platform itself. Similarly, calls to static functions in the `Kernel` class are mapped to code that actually manipulates the corresponding kernel structures and procedures.

It may be that the agent platform also needs to communicate with the agent (e.g. to notify it that it has just arrived at its migration point, or that the migration attempt failed for some reason, or to warn it that it is about to be terminated and thus should free any resources, etc.). For this purpose, any agent will expose a number of callback methods (defined in the `IAgent` interface implemented by all agents) such as `onCreated()`, `onStop()` etc., that are invoked by the platform whenever an event relevant to the agent occurs. A similar mechanism allows the agent to register callbacks with the kernel (for instance, a timer-based task, a tasklet, being notified that the system is about to suspend, etc.).

## D. JVM implementation details

1) **Class loading and representation**: The MAGIKMAIDS JVM accepts agent code in the form of a collection of inter-related `.class` files. The collection must be stand-alone and complete, i.e. a class may only reference either another in the same collection or a built-in class, and all of the classes required by the agent must be present, as new classes cannot be loaded after the JVM context has been initialized.

   After a thorough verification of the agent class, the JVM will perform an integration step whereby the constants, strings and class/field/method references in each class file are unified
and translated to much more compact and efficient internal representations. For instance, a field reference will translate to an offset in the object data block (as happens for C++, for instance), the exact layout of which can be computed during this integration phase. Method references, on the other hand, will translate to an offset in a C++-like virtual table specific to each class, so that invocations at runtime need only pass through a minimal amount of indirection to find the correct code in a given context. The method bytecode itself is also translated to a more architecture-specific representation, and we have plans to implement a JIT stage whereby equivalent native code would be generated and summarily optimized.

2) Object representation: Objects are generally represented as in most implementations of C++, i.e. as raw blocks of data containing field data at well-known offsets, plus a pointer to the virtual table corresponding to the run-time class for that object. Additionally, objects in the MAGIKMAIDS JVM also feature a pointer to the Run-Time Type Information (RTTI) for their class (which is needed by some interpretation functions), and a reference counter for use by the garbage collector. Object references are simply pointers to the raw object datablock - there is no need for double indirection as we do not use a Copy Collection garbage collector as found in modern userspace JVMs.

3) Built-in classes: The MAGIKMAIDS JVM provides special native code implementations for both MAGIKMAIDS-specific interfaces (Kernel, Magik) and typical built-in classes, such as arrays, java.lang.String, java.lang.Number descendants, java.util.List/Map/Set collections, etc. Objects belonging to such classes typically have custom memory representations and native code functionality that goes beyond the power of bytecode.

Collection classes are internally mapped to the efficient implementations already present in the Linux kernel for that particular concept, i.e. java.util.LinkedList to linked lists, java.util.TreeMap and java.util.TreeSet to red-black trees, etc.

4) Garbage collection: The MAGIKMAIDS JVM uses reference counting for garbage collection. This has the key advantages of being fast, simple, and more predictable than mark-and-sweep. Just as importantly, memory is reclaimed immediately once an object is no longer referenced.

The disadvantages of this solution include an increased cost for operations dealing with references, as well as an inability to deal with reference cycles. Agent programmers are expected to recognize and avoid such reference patterns, or include destructor-like functions that manually unlink the components in a cycle before the last reference to them is abandoned.

E. The agent simulation and compilation environment

MAGIKMAIDS features an innovative method of developing and testing agents in an accessible and productive manner. The MAGIKMAIDS kit features a Java library that contains the IAgent definition as well as mock userspace implementations of the Kernel. [...] and Magik. [...] services. Agents are compiled against this library and a series of standalone .class files are produced, that can then be fed to the agent platform so as to spawn a corresponding agent.

While in the “real” kernel, calls to the Magik and Kernel services map naturally to events in the actual kernel and agent platform LKM. However, agents can also be executed in userspace within an agent-enabled distributed system simulation environment that we have created for this purpose. In that situation, the “dummy” functions in the simulation library map to virtual calls to the simulated machine in the context of that agent instance. Thus, agent developers can use the exact same code to readily test their designs within a safe simulated environment before actually deploying the agents in the field.

V. RESULTS

A. Speed

Our first test involved estimating the execution speed for Java bytecode as achieved by the JVM implementation used in MAGIKMAIDS. The test was performed by randomly executing the handler functions for both a regular MAGIKMAIDS agent, and an equivalent agent-like LKM implemented in C. An average execution time was computed for each case after a sufficient number of runs had been performed.

Our results show that the virtualized agent executes about 40 times slower than equivalent native C code. While this may seem discouraging, one must note that an agent built according to the principles we outlined would execute most of its code in short bursts whenever its callbacks are invoked. Even with the considerable slowdown of virtualization, these callbacks still amount to only a small fraction of the code executed by the system during any given time. Agents will carry out more
intensive processing using a work queue or a kernel thread, where they can be preempted so as not to use up too much of the system resources. In conclusion, we believe the total performance impact to be minimal.

Moreover, we are in the process of implementing a JIT compilation stage as in [8]. Once it is finished, it is likely to improve the above figure greatly, perhaps by a factor of up to 10.

B. Memory usage

We next examined the memory impact of the agents and the agent platform itself. Memory efficiency is just as important at CPU efficiency as the kernel needs to leave as much space as possible for the userspace applications; however, it is worth noting that if the comparison is made with respect to a userspace agent platform, some memory is going to be used for that purpose whichever space is used (user or kernel), therefore MAGIKMAIDS only needs to meet a relative standard of efficiency, i.e. not use much more memory than an equivalent userspace platform.

Key memory footprint measurements are listed in table I. Note that the figures are estimates because the exact sizes depend on the architecture for which the module is compiled, as well as other factors (topology, state etc.).

<table>
<thead>
<tr>
<th>MAGIKMAIDS platform footprint</th>
<th>200KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LKM code size (excludes classlib native code)</td>
<td>50KB</td>
</tr>
<tr>
<td>Agent-independent state data</td>
<td>350KB</td>
</tr>
<tr>
<td>Class library (+ native code)</td>
<td>50KB</td>
</tr>
<tr>
<td>Typical agent footprint</td>
<td>10KB</td>
</tr>
<tr>
<td>Agent classes (instance-independent)</td>
<td>50KB</td>
</tr>
<tr>
<td>Per-instance state data</td>
<td>10KB</td>
</tr>
</tbody>
</table>

It can be seen that, while large for a kernel component, the memory footprints are hardly noticeable in a modern system equipped with gigabytes of RAM, and certainly much less than those typical of a Java-based userspace agent platform.

VI. CONCLUSION

We believe we have proven the feasibility of an agent platform in the space of the Linux kernel. The MAGIKMAIDS environment offers a rich API that enables the development a wide variety of monitoring and intervention tasks to be performed using mobile agents. At the same time, the CPU efficiency and memory footprint of the platform are well within acceptable limits provided minimal care is taken in the design of the agents. Finally, a major advantage of our solution is represented by the simulation environment that greatly facilitates the development and testing of agents in a productive and safe manner.

Future work may yet be performed in the direction of improving the efficiency of the JVM component of MAGIKMAIDS. An obvious first step would be the integration of a JIT compilation component akin to that in [8] - a task made easier by the fact that the current JVM already performs some key steps such as establishing the binary layout of objects and building virtual tables. Another possible approach involves making use of advanced CPU instructions specifically designed for implementing JVM operations, as many recent ARM chips feature these.

VII. ACKNOWLEDGEMENTS

This work was supported, in part, by Google, INC., University Relations, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/88/1.5/S/60203.

REFERENCES