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

Cristian Dinu
"Politehnica" University of Bucharest
Email: cristian.dinu@cti.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 nears unfeasibility, and 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*, our paper takes the novel approach of considering the implementation of such a platform in *kernel space*, dubbed MAGIKMAIDS. We begin by making a case for why a kernel space implementation would present worthwhile advantages, then we investigate possible implementation approaches and follow with the architecture and details of the Java Virtual Machine-based approach that we have selected. Finally, we show results regarding the performance, adaptability and ease of use of our mobile agent platform. Our contribution also includes a simulation environment that greatly eases the development and testing of agents for the MAGIKMAIDS platform.

I. Introduction

It would not be too inappropriate to call our time "the century of the distributed system". Indeed, 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 great 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 - and only 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 of security, safety and CPU and memory use, we believe that such a scenario would present several unique advantages that make such an undertaking worthwhile:

- 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.
- 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 dubbed 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 LKMs (Linux Kernel Modules) 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:

- 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.
- 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 .configs, 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).
- 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 represeting 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:

- 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).
- 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. It is still possible to call arbitrary kernel functions, albeit in a most unelegant way, by employing *kallsyms* trickery.

On the other hand, the virtualization layer resolves many of the problems that plagued Approach A:

- Agent portability is no longer a problem - 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.
- 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 it the most promising method was B), owing to the fact that a less than optimally efficient agent platform is still preferrable 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-Helman 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.

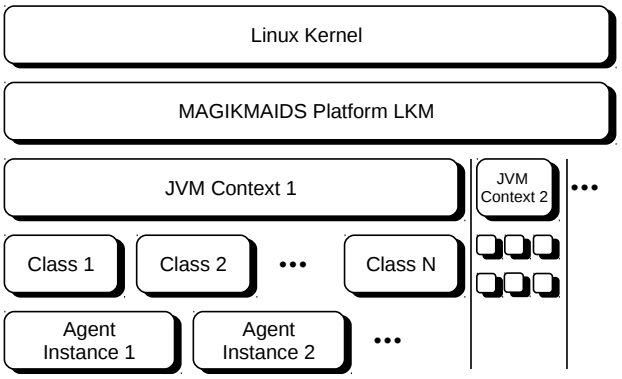

Fig. 1. The MAGIKMAIDS architecture

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 entitites 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, though, as stated previously, they cannot directly reference each other. 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 mehtods exposed by each entity. A diagram of this interaction is shown in figure 2.

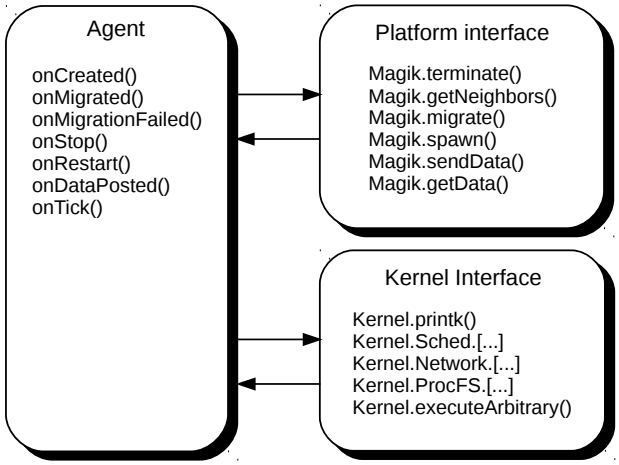

Fig. 2. Agent, kernel and platform interfaces

It can be seen that 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 to warn it that it is about to be terminated and thus should free any resources). For this purpose, any agent will expose a number of callback methods

(defined the `IAgent` interface implemented by all agents) such as `onCreated()`, `onStop()` etc., that are called 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.).

A selection of the most important callbacks and functions exposed by the three parties follows, so as to offer an idea as to the nature of the programming environment available to agents:

1) Agent platform functions:

- `Magik.terminate()`: Requests that the platform terminate this agent instance. Before the agent is actually terminated, its `onStop()` callback will be executed so that there is a uniform way of the agent freeing its associated resources.
- `Magik.getNeighbors()`: Obtains an iterator through the set of direct agent-enabled neighbors of the current machine. The agent can use this for deciding its next migration target.
- `Magik.migrate()`: Requests that the agent be migrated to a target machine of its choice. The platform will stop the agent and provide it with an opportunity to save its data in serialized form, then send it to the destination in a packet containing the agent .class files and the serialized data. If the agent arrives intact and is accepted by the target, it will be restarted in its new environment and provided with the saved state data so that it can restore its state.
  The agent platforms use confirmation messages to verify that an agent has actually successfully migrated to the other side. Should an agent be corrupted in transit or rejected, the migration will be retried a number of times and aborted if these are unsuccessful as well. Following an aborted migration, the agent will be restarted on its original machine and notified of the situation so that it can perform error recovery.
- `Magik.spawn()`: Spawns (creates) a new instance of any desired agent type, which is then fed initialization data specified by the calling agent. This powerful function can be used as the basis of many interesting execution patterns, such as viral replication, master-slave relationships, etc. Access to this function is tightly controlled, with most agents being restricted with respect to the type and number - if any - they are allowed to spawn.
- `Magik.sendData()`: Posts a message (in serialized format) for any interested agents on the current node. Messages may be *transitory*, in which case they are immediately delivered to other running agents (which can ignore them or take action), or *persistent*, in which case they are stored in the environment and can be retrieved later by agents that have knowledge of their existence.
- `Magik.getData()`: Queries the agent platform for any persistent messages left by a previous agent (possibly ourselves), optionally removing the message if it is recognized. The `sendData()` and `getData()` functions working in tandem allow a cooperation effect somewhat similar to that found in the venerable JavaSpaces paradigm.

2) Agent callbacks:

- `onCreated()`: This is called when an agent has just been spawned into the MAGIKMAIDS environment and provides it with the opportunity to perform initialization of its state. The agent also receives a `DataInputStream` object from which it can read any initialization data provided by its creator.
- `onStop()`: This is called when the agent is about to be stopped for any reason (i.e. before migration, termination, hibernation etc.) and provides it with an opportunity to save its serialized state data to a `DataOutputStream` provided by the agent platform. Before stopping, the agent will also have to free any resources it holds (memory, handles) and stop any active timers.
- `onRestart()`: This is the converse function that is called when an agent is restarted and allows it to reload its state from a provided `DataInputStream`, as well as restart timers and reacquire resources.
- `onMigrated()`: This is called after an agent has successfully migrated to a new platform and should contain code directing the next actions of the agent.
- `onMigrationFailed()`: This is called when an agent could not be migrated successfully to its intended destination (due to network problems, the destination rejecting the agent etc.). The agent will remain on its current machine and execute code in this callback so as to recover from the error.
- `onDataPosted()`: This is called when a new message has been broadcast by an agent on the current platform. The agent can inspect the message using a `DataInputStream` and take action should it find it relevant.

3) Kernel functions:

- `Kernel.printk()`: Prints formatted messages to the console. The severity parameter present in the original `printk()` function is also implemented.
- `Kernel.executeArbitrary()`: executes an arbitrary kernel function given by name and parameter signature. This dangerous but very powerful function internally makes use of the `kallsyms` kernel facility whereby the kernel stores a table containing the name of every public symbol and its address so that it can be looked up at runtime.
- `Kernel.Work` and `Kernel.Timer`: these classes represent interfaces to the workqueue and timer callback mechanisms in the Linux kernel and are designed for use in a manner similar to `java.util.Thread` and `java.util.Timer` respectively.
- `Kernel.Sched.[...]`: a 'namespace' for all functions related to the scheduler subsystem in the Linux kernel
- `Kernel.Network.[...]`: a 'namespace' for all

functions related to the networking subsystem in the Linux kernel
- `Kernel.ProcFS.[...]`: a 'namespace' for all functions related to the procFS subsystem in the Linux kernel

D. JVM implementation details

1) Class loading and representation: The MAGIKMAIDS JVM accepts agent code in the form of a collection of interrelated .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. The JVM will perform a summary verification of the agent class files and perform an integration step whereby the constants (scalars, strings, class/method references) in each class file are merged into a unified index and linked to the built-in implementations.

The resulting internal class representation is far more compact and easier to parse and execute than the aggregated class files. Redundant information is eliminated, methods are referenced by offsets in the object data block and virtual table respectively instead of strings (in fact, class/method names are not stored at all, as the JVM does not provide reflection facilities), and the method code is pasted into a unified address space.

The internal class representation also contains vtables (virtual function tables) for each class, as well as a map of every location where a reference may be stored in that particular object (for use by the garbage collector). User-supplied functions are generally implemented in bytecode and referenced by a pointer in the bytecode array, while methods in the built-in classes (`System`, `Magik`, etc.) are implemented using native code and may feature special behavior.

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 RTTI block of the class the object belongs to, where important information such as the vtable may be accessed. Object references are direct pointers to such data blocks - no double indirection is needed as in most modern JVM implementations due to the use of a simple garbage collector as opposed to a Copy-Collection variant.

It is obvious that this representation is highly compact and has only minimal overhead with respect to the C structures used in the kernel.

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 garbage collector used in the MAGIKMAIDS JVM is of the mark-and-sweep variety. It is worth noting that the garbage collector is only invoked when:
- Control is transferred between the agent and the agent platform (i.e. at the end of a callback) or
- The agent explicitly requests garbage collection

Thus, agents can execute code without having to worry that they will be interrupted in the middle of an important operation by a stop-the-world garbage colllection process. The fact that each agent has a private heap also helps minimize the performance impact of garbage collection.

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.

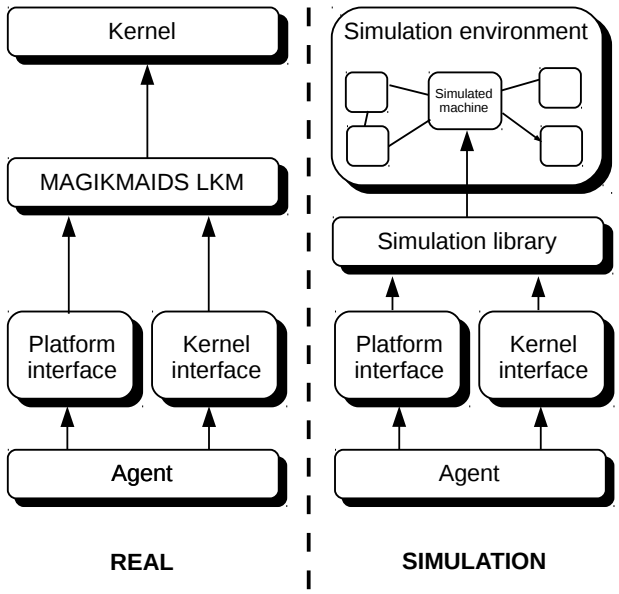

Fig. 3. Interfacing with the simulated environment

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" implemented as a LKM in C. An average execution time was computed for each case after a sufficient number of runs had been performed. Additionally, we computed an "extrapolated JIT performance" by which we mean the performance that would likely be obtained if we implemented a JIT stage as in [8]. Finally, all values were normalized with respect to the fastest time (corresponding to the native LKM solution) so that the relative performance impact of interpretation and virtualization is easily apparent.

The results are shown in table I.

Approach	Rel.time
Traditional LKM (native)	1x
MAGIKMAIDS JVM (interpreted)	18.6x
MAGIKMAIDS JVM+JIT (estimation)	1.8x

TABLE I
MAGIKMAIDS execution time

The one-order-of-magnitude difference between interpreted code in the MAGIKMAIDS JVM and native code may seem worrying, but it is worth noting that compliant agents will only execute short sections of code in the callbacks themselves (which are blocking). 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. Additionally, one can see that by adding a JIT stage to the platform, one may substantially increase performance up to a level close to that of native code, such that the differences are hardly noticeable.

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 user space 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 II. 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.).

MAGIKMAIDS platform footprint	
LKM code size (excludes classlib native code)	200KB
Agent-independent state data	50KB
Class library (+ native code)	350KB
Typical agent footprint	
Agent classes (instance-independent)	50KB
Per-instance state data	10KB

TABLE II
Platform memory usage

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.

C. Productivity

One final test of our platform involved estimating the productivity benefits introduced by MAGIKMAIDS userspace testbed for agents. The same kind of agent - a topology discovery agent that hopped from machine to machine and returned with a map of the entire system - was implemented using two paradigms: first, that of a traditional LKM written in C, which directly called internal MAGIKMAIDS functions to achieve migration and was tested and debugged in a virtual machine; and second, that of a MAGIKMAIDS agent written in Java and debugged using the testbed provided by MAGIKMAIDS. Key metrics measured were: the average number of bugs per thousand lines of code (bugs/KLOC), the estimated first-draft development time, and the estimated debug time (from first draft to apparently bug-free solution).

The results are shown in table III.

Approach	Bugs/KLOC	Dev time	Dbg time
Traditional LKM	17.1	1h30m	30m
MAGIKMAIDS agent	8.8	20m	5m

TABLE III
Productivity test

While this test is more subjective than the others, we believe the relative benefits of the MAGIKMAIDS development approach to be evident in the presented data.

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 enough API to enable a limitless 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.

REFERENCES

[1] D. Nessett, "Massively distributed systems: Design issues and challenges," in *Proceedings of the Workshop on Embedded Systems on Workshop on Embedded Systems*. USENIX Association, 1999, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267143

[2] J. Ahn, "Fault-tolerant Mobile Agent-based Monitoring Mechanism for Highly Dynamic Distributed Networks," *IJCSI International Journal of Computer Science*, vol. 7, no. 7, p. 3, 2010. [Online]. Available: http://ijcsi.org/papers/7-3-3-1-7.pdf

[3] S. Ilarri, E. Mena, and a. Illarramendi, "Using cooperative mobile agents to monitor distributed and dynamic environments," *Information Sciences*, vol. 178, no. 9, pp. 2105–2127, May 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S00200250700583X

[4] P. Mell, M. McLarnon, and MD., "Mobile agent attack resistant distributed hierarchical intrusion detection systems," *Information Systems Security*, no. A294193, pp. 1–8, 1999.

[5] P. S. Mandal and K. Mukhopadhyaya, "Checkpointing Using Mobile Agents in Distributed Systems," in *2007 International Conference on Computing: Theory and Applications (ICCTA'07)*. IEEE, Mar. 2007, pp. 39–45. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4127340

[6] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proceedings of the second USENIX symposium on Operating systems design and implementation - OSDI '96*, vol. 30, no. SI. New York, New York, USA: ACM Press, 1996, pp. 229–243. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.6054 http://portal.acm.org/citation.cfm?doid=238721.238781

[7] W. Jansen and T. Karygiannis, "Mobile Agent Security," *NIST Special Publication*, vol. 14, no. 5, pp. 211–218, 2000.

[8] T. Okumura and B. Childers, "Running a Java VM inside an operating system kernel," *Proceedings of the fourth ACM*, pp. 161–169, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1346279