SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 1–11. `http://www.scpe.org`

ISSN 1895-1767
© 2006 SWPS

SECURITY RISKS IN JAVA-BASED MOBILE CODE SYSTEMS

WALTER BINDER[*] AND VOLKER ROTH[†]

Abstract. Java is the predominant language for mobile agent systems, both for implementing mobile agent execution environments and for writing mobile agent applications. This is due to inherent support for code mobility by means of dynamic class loading and separable class name spaces, as well as a number of security properties, such as language safety and access control by means of stack introspection. However, serious questions must be raised whether Java is actually up to the task of providing a secure execution environment for mobile agents. At the time of writing, it has neither resource control nor proper application separation. In this article we take an in-depth look at Java as a foundation for secure mobile agent systems.

Key words. Java, JVM, mobile agents, security, process isolation, resource management

1. Introduction. The proliferation of the Java programming language [18] led to the development of numerous mobile agent platforms. Actually, Java seems perfect for developing an execution environment for mobile agents, because Java offers many features that ease its implementation and deployment. Java runtime systems are available for most hardware platforms and operating systems. Therefore, mobile agent platforms that are built on Java are highly portable and run seamlessly on heterogeneous systems. Furthermore, mobile agents profit from continuous performance and scalability enhancements, such as increasingly sophisticated compilation techniques and other optimizations, which are provided by the underlying Java Virtual Machine (JVM) [23].

In addition to portable code, Java offers a *serialization* mechanism allowing to capture a mobile agent's object instance graph before it migrates to a different host, and to resurrect the agent in the new environment. Java also supports dynamic loading and linking of code by means of a hierarchy of *class loaders*. A class loader constitutes a separate *name space* that can be used to isolate classes of the agent system and of different agents from each other.

In general, mobile agent platforms execute multiple agents and service components concurrently in a time sharing fashion. Java caters for this need by means of multi-threading. Java is also a *safe* language, which means that the execution of programs proceeds strictly according to the language semantics (this is not entirely true, as we discuss in Section 2.3). For instance, types are not misinterpreted and data is not mistaken for executable code. The safety properties of Java depend on techniques such as *bytecode verification*, *strong typing*, *automatic memory management*, *dynamic bound checks*, and *exception handlers*. On top of that, the Java 2 platform includes a sophisticated security model with flexible access control based on dynamic stack introspection.

In summary, Java is highly portable and provides easy code mobility. This caused numerous mobile agent systems based on Java being developed and experimented with. From the point of security, two points can still be criticized: first, all systems focus on a particular aspect of agent mobility and none address all problems simultaneously, whose solution is required to come up with a system ready for field use. This is particularly true for security. Second, practical experience with Java shows that considerable security concerns remain, which are discussed in the next section.

This article, which is based on reference [8], is structured as follows: Section 2 discusses a series of shortcomings of current Java environments that reduce system stability in the presence of erroneous code and may be easily exploited for attacks by malicious mobile agents. These flaws can be overcome by a combination of isolation of mobile agents executing in a single JVM and resource control. While current standard JVM implementations do not address these issues, research has matured and several solutions have been suggested, also within the Java Community Process (JCP). Section 3 deals with issues of mobile agent isolation, whereas Section 4 addresses resource control. In both of these sections we compare the strengths and limitations of different approaches. Finally, the last section concludes this article.

2. Java Security Problems. In this section we give an overview of shortcomings in the JVM that hinder the development of secure and reliable mobile agent platforms. We discuss problems in the area of object management (Section 2.1), thread management (Section 2.2), bytecode verification (Section 2.3), as well as shortcomings of the Java security model (Section 2.4) and the lack of resource control (Section 2.5).

2.1. Object Management. A Java class is represented in the JVM by a *class object*. The class object that represents a class is initialized upon the first *active* use of that class. Mere declaration of a typed variable does not constitute active use and does not trigger initialization of the class that represents that type. However, as soon as a static method actually declared in that class is invoked, or a constructor is invoked, or a non-constant field is accessed, the class is initialized.

[*]Faculty of Informatics, University of Lugano, Lugano, Switzerland, `walter.binder@unisi.ch`
[†]FX Palo Alto Laboratory, Inc., 3400 Hillview Ave, Bldg 4, Palo Alto, CA, USA, `volker.roth@acm.org`

LISTING 1
Signature for a callback in the Serialization Framework.

```
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

LISTING 2
Example DoS attack on thread creation.

```
synchronized(Thread.class) {
  while (true);
}
```

When a Java class is initialized, first its class variable initializers and static initializers are executed. This opens a loophole for executing potentially malicious code before even the first instance is generated.

Further loopholes are hidden in the *Serialization Framework* of Java. During deserialization of an object instance, no constructors are invoked. The fields of unmarshalled objects are initialized directly. However, if the object that is unmarshalled implements a method of the exact signature given in listing 1 then this method is invoked in the deserialization process. A similar callback method exists for object serialization. This means that:

- agent state appraisal and authentication must complete before the agent instance is marshalled and the first agent class is loaded into the VM. Once this happens, it is already too late to defend against malicious agents.
- whenever an agent class is initialized, or an agent instance is marshalled or unmarshalled, the agent may take over the current thread and perform a variety of unexpected operations.

The loopholes described above potentially allow agents to run code before the system is prepared for it.

Another loophole in Java's garbage collector allows to "plant an egg" that is hatched after the agent has been terminated. When the VM detects that no strong references exist anymore to some object then it garbage collects this object and reclaims the memory occupied by the object. However, prior to that the garbage collector gives the object an opportunity to clean up any leftover state by invoking the finalizer of that object (if it is implemented). Consequently, if the method call does not terminate then no garbage is collected anymore and the VM eventually crashes from a lack of memory.

A less obvious attack would be to set an alarm (by means of a new thread) that triggers a destructive method only after a delay. In that case, the log files of the agent system (if there are any) show that the malicious agent already left the server and thus cannot possibly be responsible for the crash. At the very least, it becomes complicated to determine what actually happened and to prove it to someone else.

The developer of an agent system might be tempted to eliminate these loopholes by refusing to load any classes that implement the finalizer method. However, this is insufficient because several classes in the Java core packages already implement a finalizer and invoke additional callbacks in it. A malicious agent might, for instance, provide a class that inherits e.g., from *FileInputStream, FileOutputStream,* or *ZipFile*.

Any of these classes invokes method *close()* in its finalizer. Hence, rather than overriding the finalizer itself, malicious code may override the *close()* method. Consequently, all classes that inherit from one of these must be blocked as well (at least, if they implement *close()*).

Regardless how an agent becomes executed, once it runs it may hamper other threads and agents in a variety of ways. One option to launch a *denial of service* (DoS) attack is to synchronize on class locks. In Java all class objects of classes loaded by the system class loader are visible and some classes synchronize on their class locks. A simple DoS attack is illustrated in listing 2. If executed, no new threads can be generated because any attempt to increase the thread counter will lead to a deadlock situation (see also the relevant source code excerpt from the JDK 1.6 given in listing 3). Class locks can also be used to implement covert channels [22].

Clearly, touching an object is a dangerous thing. Yet, mobile agent systems often allow uncontrolled aliasing (sharing of object references), which is both convenient and typical of object-oriented programming. Again, DoS attacks can take on various forms. Catching the current thread is one possibility, keeping references to other agents' objects is another. As long as a strong reference to some object exists, it will not be garbage collected. In Java it is not possible to revoke an object reference. However, dynamic proxy generation mechanisms (which are available since JDK version 1.3) can be applied to this problem. This approach is taken e.g., in the SeMoA mobile agent system.[1]

[1] `http://www.semoa.org`

LISTING 3
Vulnerable code in the JDK 1.6.

```
private static long threadSeqNumber;

private static synchronized long nextThreadID() {
  return ++threadSeqNumber;
}
```

In summary, the concurrent execution of multiple agents requires isolation boundaries, where the passing of references has to be controlled. Marshalling and unmarshalling of objects must be done by a thread that can be sacrificed, or belongs already to a sandbox that is set up in advance for the object in question. Migration must take place only after all threads of a mobile agent have terminated and none of its classes is on the stack of any running thread anymore (or referenced by any object with a strong reference). Abuse of class locks is a matter that is addressed best by means of a modification of Java.

2.2. Thread Management. The Java language includes a set of APIs and primitives to manage multiple concurrent threads within a Java program. Synchronization between threads is based on *monitors*, which are associated with objects. Java *synchronized*{} statements are mapped to matching *monitorenter* and *monitorexit* instructions at the bytecode level. Monitors are implemented based on *locks*; each object has an associated lock that is used whenever a *synchronized*{} statement refers to that object. Methods of an object can be declared *synchronized*, which implies that the method is executed in a monitor whose lock is the one associated with that object. Instance methods are associated with the lock of the object instance, whereas static methods are associated with the lock of the object instance that represents the object class (and which is of type *java.lang.Class*).

2.2.1. Inconsistency due to Asynchronous Termination. One important function of a mobile agent platform is the termination of agents. When an agent migrates or terminates, all of its allocated resources should be reclaimed as soon as possible. That is, all threads of the agent shall be stopped and memory allocated by the agent shall become eligible for garbage collection. Especially when a misbehaving agent is detected it has to be removed from the system with immediate effect.

Java allows to asynchronously terminate a running thread by means of the *stop* method of class *java.lang.Thread*. This method causes a *ThreadDeath* exception to be thrown asynchronously in the thread to be stopped. Unfortunately, thread termination in Java is an inherently unsafe operation, because the terminated thread immediately releases all monitors. Consequently, objects may be left in an inconsistent state. As long as these objects are exclusively managed by the agent to be removed from the system, a resulting inconsistency may be acceptable.[2]

However, if a thread is allowed to cross agent boundaries for communication purpose (e.g., inter-agent method invocation), the termination of a thread has to be deferred until it has completed executing in the context of other agents. Otherwise, the termination of one agent may damage a different agent that is still running in the system. Unfortunately, delayed thread termination prevents immediate memory reclamation, because references to objects of the terminated agent may be kept alive on the execution stack of the thread. Even worse, if shared objects, such as certain internals of the JVM, are left inconsistent, asynchronous thread termination may result in a crash of the JVM. For this reason, the *stop* operation has been deprecated in the Java 2 platform.

To solve these problems, the mobile agent platform has to enforce a thread model where each thread is bound to a single agent. Threads must not be allowed to cross agent boundaries arbitrarily. Upon the invocation of a method in a different agent, a thread switch is necessary. The called agent has to maintain worker threads to accept external method calls. However, this approach negatively affects performance, because thread switches are rather expensive operations.

To ensure the integrity of shared data structures and of JVM internals, the mobile agent system has to enforce a user/kernel boundary, where shared structures are manipulated only within the kernel. With the aid of a locking mechanism, it is possible to ensure atomic kernel operations. That is, requests for asynchronous termination are deferred until the thread to be stopped has left the kernel. Because kernel operations can be implemented with a short and bounded execution time, domain termination cannot be delayed arbitrarily. All critical JVM operations have to be guarded by a kernel entry. Again, this solution causes some overhead.

[2]If the agent state is captured (e.g., serialized) after termination, inconsistencies may corrupt the further execution of the agent on other platforms. The agent is responsible to freeze its non-transient state before requesting migration.

LISTING 4
A method to prevent thread termination.

```
while (true) {
  try {
    while (true);
  }
  catch (Throwable t) {}
}
```

LISTING 5
Catching ThreadDeath *can be prevented by rewriting the bytecode in listing 4 in a way that is functionally equivalent to the given Java code transformation.*

```
while (true) {
    try {
        while (true);
    }
    catch (Throwable t) {
        if (t instanceof ThreadDeath) {
          throw t;
        }
    }
}
```

2.2.2. Interception of Asynchronous Termination. There are further problems with asynchronous thread termination: The *stop* method does not guarantee that the thread to be killed really terminates, because the thread may intercept the *ThreadDeath* exception. For instance, consider the code fragment in listing 4, which cannot be terminated easily.

Note that not only exception handlers may intercept *ThreadDeath* exceptions, but *finally*{} clauses may prevent termination as well. However, the Java compiler maps *finally*{} statements to special exception handlers. Thus, it is sufficient to solve the problem with exception handlers that catch *ThreadDeath* or a superclass thereof.

The JavaSeal mobile agent kernel [10] enforces a set of restrictions on exception handlers that may catch *ThreadDeath*, in order to ensure the termination of such handlers. However, this approach imposes severe restrictions on the programming model. For instance, untrusted agents may not use *finally*{} clauses. Furthermore, the JavaSeal implementation is incomplete, as a *monitorexit* instruction[3] in an exception handler may throw *NullPointerException* or *IllegalMonitorStateException*, which can be caught by user code.

Another solution to this problem involves rewriting of agent bytecode so that *ThreadDeath* exceptions are immediately thrown again by all exception handlers. This approach is used in the J-SEAL2 mobile agent kernel [3]. Listing 4 shows a portion of Java code and listing 5 its rewritten counterpart. For the ease of reading, we give the transformation at the Java level, whereas rewriting would be done actually at the JVM bytecode level.

2.2.3. Undefined Thread Scheduling. Neither the Java language [18] nor the JVM specification [23] define the scheduling of Java threads. Therefore, it is not guaranteed that, on every Java platform, high-priority surveillance threads preempt other threads. A related problem is priority inversion. This means that a high-priority thread may have to wait until a low-priority thread releases a monitor. However, the low-priority thread may not be scheduled if there are other threads ready to run that have a higher priority.

Most standard Java runtime systems offer native threads that are scheduled by the operating system i. e., the scheduling is platform-dependent. This means that a surveillance task using a high-priority thread that has been tested in one environment may not work well in another one, contradicting the Java motto “write once, run anywhere.” Moreover, an increasing system load (an increasing number of threads) often affects the scheduling and may prevent high-priority threads from executing regularly.

Priority inversion may be addressed by temporarily raising the priority of a thread executing a critical section. However, in many JVMs adapting thread priorities is an expensive operation, since it triggers the scheduler.

The realtime specification for Java [9] specifies priority-based preemptive scheduling with at least 28 different levels of priority for all compliant implementations. The realtime specification covers many other topics important for realtime

[3]The compilation of a *synchronized*{} statement creates an exception handler whose task is to release the monitor in case of any exception. Because synchronization is an important concept of the Java language, JavaSeal allows agents to use the *monitorexit* instruction within exception handlers.

LISTING 6
Example bytecode that acquires a lock that is not released after completion.

```
static void captureMonitor(java.lang.Class)
   0 aload_0
   1 monitorenter
   2 return
```

systems. Therefore, standard Java runtime systems will not likely conform to the realtime specification, because they target environments without realtime requirements and the underlying operating system does not necessarily offer realtime guarantees either. Consequently, we think that a subset of the realtime specification should be integrated into a new version of the standard Java specification, so that applications that depend on scheduling mechanisms – such as mobile agent systems – may run consistently across different JVM implementations.

2.3. Bytecode Verification. Java relies on static and dynamic checks to ensure that the execution of programs proceeds according to the language semantics. Before a program is linked into the JVM, the Java bytecode verifier performs static analysis of the program to make sure that the bytecode actually represents a valid Java program. Dynamic checks (e.g., array bounds checks) are incorportated in many JVM instructions.

Unfortunately, bytecode verifiers of several current standard Java implementations also accept bytecode that does not represent a valid Java program. The result of the execution of such bytecode is undefined, and it may even compromise the integrity of the Java runtime system.

At the Java bytecode level, the allocation of an object is separated from its initialization. One important task of the Java verifier is to prevent uninitialized objects from being used (e.g., the fields of uninitialized objects must not be accessed and only a constructor can be invoked on an uninitialized object). In [14] the authors take an in-depth look at object initialization in Java and define rules to be enforced by the Java verifier. However, one particular issue is not addressed: finalizers will be invoked on uninitialized objects, which undermines the properties of object initialization that are meant to be enforced by the Java verifier.

Another source of problems are the synchronization primitives of the Java bytecode. The example given in listing 6 depicts the bytecode of a method that acquires a class lock without releasing the lock after completion:

This code sequence does not constitute a valid Java program because the *monitorenter* instruction (which acquires a lock) is not paired with a matching *monitorexit* (which releases the lock). Neither is an exception handler present that releases the lock in the case of an exception. Nonetheless, several Java verifier implementations do not reject this code. The effects of executing this code are undefined and depend on the particular JVM implementation. We tested the method invocation *captureMonitor(Thread.class)* with 3 different JVMs on a Windows platform and observed varying outcomes with each of them:

Hotspot Server VM 2.0: An *IllegalMonitorStateException* is thrown at the end of the method and the monitor is released.

JDK 1.4/1.5 (Hotspot server and client): An *IllegalMonitorStateException* is thrown at the end of the method.

JDK 1.2.2 Classic VM: No exception is thrown and the monitor remains locked until the thread that has acquired the monitor terminates.

IBM JDK 1.3.0 Classic VM: The monitor is not released even after the locking thread has terminated. Subsequent attempts by other threads to create new threads are blocked, because thread creation involves a static synchronized method, which waits for the release of the class lock. In fact, this kind of attack is similar to the DoS attack shown in Section 2.1. However, this attack is even worse, because the lock is not released after all attacker threads have terminated, whereas the attack in Section 2.1 can be resolved by stopping the attacking thread.

Listing 7 depicts another example of disarranged bytecode that is not rejected by several standard Java verifiers. In this bytecode sample, the target of the exception handler is the first instruction protected by the same handler. Such a construction is not possible at the Java language level, because a *try*{} block cannot serve as its own *catch*{} clause.

At bytecode position 1 there is an infinite loop (*goto 1*), which is protected by the exception handler.[4] In case of an exception, the handler continues the same loop. Therefore, it is not possible to stop a thread executing such code. Even the transformation shown in listing 5 does not help, since its application would cause an infinite loop of catching and re-throwing the same *ThreadDeath* exception.

[4]The *aconst_null* instruction at position 0 ensures that there is always a single reference on the stack at code position 1 (this constraint is enforced by the Java verifier). When an exception is caught, the stack is cleared and a reference to the exception object is pushed onto the stack. The *return* instruction is never reached.

LISTING 7
Example bytecode whith a disarranged exception handler.

```
static void preventTermination()
   0 aconst_null
   1 goto 1
   4 return

Exception table:
   from   to  target type
     1     4     1   <Class java.lang.Throwable>
```

In order to prevent such attacks, improved bytecode verification is necessary. A better solution would be the definition of an alternative Java class format, which enables simpler verification. For instance, *Slim Binaries* [17] encode the abstract syntax tree of a program and can be verified easily, because the code can be restricted to valid syntax trees of the programming language. Thus, expensive bytecode verification can be avoided. This is particularly beneficial for mobile agents, whose startup overhead frequently exceeds execution time before migration.

2.4. Security Model. One of the most prominent security features in the Java security model is the fact that permissions of a thread are limited to the permissions granted to the least privileged class on its execution stack. Permissions are assigned by the *class loader* when the class is defined. Any class can check whether the current thread has a particular permission, by invoking the *access controller* with a template of the permission that shall be checked. The access controller responds by throwing an exeception if the permission is not granted, and silently returns otherwise.

Classes can execute *privileged actions*, which means that the class assumes responsibility for subsequent actions, and the permissions granted subsequently to the executing thread shall be the ones granted to the class that invoked the privileged action (in that case, stack introspection stops at the privileged context). New permission types can be introduced by means of a permission class that represents the type. While this gives great flexibility in terms of implementing security checks it also lacks central control.

Security checks as well as privileged actions may be scattered throughout the class packages, and it is next to impossible to determine with certainty whether a given application actually enforces a particular security policy. Even a small error can have disastrous effects on the system security as a whole; in particular, multiple small errors culminate into bigger ones. For instance, write access to the VM binary or permission to execute native code is virtually equivalent with granting the *all permission*.

Recall that local classes are visible globally. For instance, assume that a *logger* class writes log entries to a file. We assume that the class is initialized with a file name, and that it is allowed to write arbitrary files. Log events are potentially caused by threads with a trust level that is lower than the trust level of the logger, therefore the logger uses a privileged action to write to the log file. Since the logger class is globally visible and is initialized with a file name, any code (including code without file access permissions) may instantiate the class with an arbitrary file name and use it to write log entries to it.

Rather than binding permissions to a class, permissions need to be bound to a particular *instance*. This can be achieved as follows: in its constructors, the trusted class stores a snapshot of the current access control context (ACC) in a private variable. Whenever the privileged action is executed, the stored ACC is set. The effective set of permissions granted to the thread is therefore the intersection of the privileges granted to the trusted class and the permissions current at the time when the class was created. Consequently, less privileged code will gain no additional permissions by instantiating the trusted class, whereas the trusted *instance* can be used without restrictions. However, this is a strict design requirement and must be enforced consistently throughout the design and implementation phase.

A feature that is desirable for any mobile agent system is instant revocation of permissions. In other words, permissions granted to an agent can be expanded or withdrawn dynamically (e.g., when the agent misbehaves). One way to achieve this is to implement a custom *ProtectionDomain* that supports that feature, and which is assigned to all classes of the agent by means of a custom class loader. However, this approach is not guaranteed to work because protection domains may be compressed as a consequence of optimizations (although it does work as intended in the reference implementation of the JDK as of version 1.3).

Starting with version 1.3, the JDK provides the *DomainCombiner* mechanism that could be applied to permission revocation. However, using this mechanism is tricky. Once a permission is assigned statically to a malicious class by means of a protection domain, this permission cannot be revoked even with a *DomainCombiner*. The malicious class can always issue a privileged action with its given permissions which blocks invocation of the *DomainCombiner* further down the stack. The solution is to grant no permissions a priori, but only dynamically by means of a *DomainCombiner*.

On the other hand, if code shall have access to reserved resources when being invoked on behalf of theads of other logical entities, then that code needs to save an ACC in its original thread, and use it in a privileged action. Otherwise, the privileged action relinquishes all permissions (because none were granted statically) and its *DomainCombiner* will not be invoked on permission checks.

In summary, the security model of the Java 2 platform is very flexible and powerful on the one hand, but on the other hand it is also very complicated and depends on the perfect orchestration of all components of the application and the mobile agent middleware. This constitutes considerable risk, because breach of security or integrity of the VM may expose the account under whose authority the VM is executed.

2.5. Lack of Resource Control. Current standard JVMs do not support resource control e.g., mobile agents may spawn an arbitrary number of threads and each thread may consume an arbitrary number of CPU cycles and allocate memory until an `OutOfMemoryError` is thrown.

The lack of resource management features makes it easy to launch DoS attacks. Moreover, even in the absence of an attack, lack of awareness of the resource consumption of executing mobile agents may lead to an overload of the system and negatively impacts the system's stability.

3. Mobile Agent Isolation. In this section we argue for a strong isolation of mobile agents that execute within the same JVM, and we discuss several approaches to accomplish this goal.

3.1. Requirements. If multiple mobile agents execute within the same JVM process, faults within one agent shall not affect the stability of the JVM and of other agents i. e., strong isolation of agents is needed [4].

Language safety in Java (a combination of strong typing, memory protection, automatic memory management, and bytecode verification) already guarantees some basic protection, as it is not possible to forge object references [31]. However, language safety itself does not guarantee isolation of mobile agents. Pervasive aliasing in object-oriented languages leads to a situation where it is impossible to determine which objects belong to a certain agent and therefore to check whether an access to a particular object is permitted or not. Thus, it is crucial to introduce the concept of isolation in the JVM, similar to the process abstraction in operating systems.

What is needed is an isolation boundary around each mobile agent, which encapsulates the set of classes required by the agent, its threads, as well as all objects allocated by these threads. Different mobile agents must not share any structures that could cause unwanted side effects, such as class locks (see Section 2.1).

As we have seen in Section 2.2.1, threads crossing mobile agent boundaries hamper the safe termination of agents. To solve this problem, a thread model is needed where each thread is bound to a single agent. Threads must not be allowed to cross agent boundaries arbitrarily. Upon the invocation of a method in a different agent, a thread switch is necessary. The called agent has to maintain worker threads to accept requests by other agents. Messages must not be passed by reference between agents, since this would create inter-agent aliasing. However, this approach may negatively affect performance, because of the extra overhead due to thread switches.

Closely related to mobile agent isolation is the safe termination of agents. If there is no sharing between agents and threads cannot cross agent boundaries, the removal of an agent will not leave other agents in an inconsistent state. For instance the execution platform does not need to care about potential inconsistencies and instead may simply terminate all threads running in an agent in a brute force fashion. Still, as shown in Section 2.2.2, the `stop()` method of `java.lang.Thread` is not suited for this purpose. Hence, a dedicated primitive for agent termination is necessary. Mobile agent isolation itself does not solve the problems of bytecode verification discussed in Section 2.3, but it confines them to the faulty agents, which can be safely terminated.

3.2. Solutions. At a high level, we distinguish between solutions that aim at providing isolation within standard Java runtime systems and approaches that rely on a modified JVM.

Among the first systems that added some sort of process model to standard JVMs were J-Kernel [29] and Java-Seal [27]. Both of these systems were implemented in pure Java. Hence they are available on any standard JVM.

3.2.1. J-Kernel. J-Kernel [29] is a Java micro-kernel supporting isolated components. In J-Kernel communication is based on capabilities. Java objects can be shared indirectly between components by passing references to capability objects. However, in J-Kernel inter-component calls may block infinitely and may delay component termination. Moreover, the classes of the JDK are used by all components, which may introduce some unwanted side-effects between components. Consequently, J-Kernel offers only incomplete isolation.

3.2.2. JavaSeal and J-SEAL2. JavaSeal [27] was designed as a secure system to execute untrusted, mobile code. It supports the hierarchical process model of the Seal calculus [28], where isolated components are organized in a tree. Each component has its separate set of threads, which are not allowed to cross component boundaries. In contrast to J-Kernel, component termination in JavaSeal cannot be delayed by blocked threads.

JavaSeal allows only for very restrictive communication between components that are in a direct parent-child relationship. This allows policy components to be inserted in the hierarchy in order to control all communication of a child component. If a message has to be transmitted between components that are not direct neighbours in the hierarchy, it must be routed along the edges of the component tree, where all involved components have to actively forward the message. Communication requires deep copies of the objects to be passed between components. As the generation of deep copies is based on the serialization and de-serialization of object graphs, the communication overhead can be excessive when compared to direct method invocation. This problem is even more severe if a message is routed through multiple components.

J-SEAL2 [3] builds on JavaSeal, but offers several improvements. The communication model supports so-called "external references," which allow indirect sharing between components. "External references" allow to shortcut communication paths in the hierarchy. They are similar to capabilities in J-Kernel, but are implemented in such a way that component termination cannot be delayed. Other improvements of J-SEAL2 include extended bytecode verification, which prevents problems as outlined in Section 2.3. Moreover, J-SEAL2 supports resource management, which will be discussed in Section 4.2.2.

In contrast to J-Kernel, JavaSeal and J-SEAL2 do not allow untrusted mobile agents to access arbitrary JDK classes, in order to prevent side-effects between components. Only a few methods of some core classes (such as `java.lang.Object`, `java.lang.String`, etc.) may be called by untrusted components. For other features (e.g., network access), dedicated, trusted service components have to be installed to mediate access to these features.

As a result, JavaSeal and J-SEAL2 offer the best possible isolation achievable on standard Java runtime systems. However, this comes at a very high price: As most of the JDK functionality cannot be directly accessed, components must be manually rewritten to use service components instead. Hence, this approach does not allow to run untrusted legacy code and it severely changes the programming model Java developers are familiar with. For this reason, such an approach to enforce strict isolation on standard JVMs does not appeal to most developers.

3.2.3. KaffeOS. The other approach is to modify a given JVM in order to support component isolation. For instance, the Utah Flux Research Group has worked on the development of specialized Java runtime systems supporting component isolation [2, 1].

Their most prominent system, KaffeOS [1], is a modified version of the freely available Kaffe virtual machine [30]. KaffeOS supports the operating system abstraction of processes to isolate components from each other, as if they were run on their own JVM. The advantage of this approach is that full component isolation is achieved without compromising the Java programming model. However, there are also severe drawbacks to this solution: The modified JVM is available only on a limited number of platforms and the costs for porting it to other environments are high. Moreover, optimizations found in standard JVMs are not available for the Kaffe virtual machine, resulting in inferior performance. In [1] the authors reported that IBM's JVM [24] was 2–5 times faster than KaffeOS. Since then, the performance of standard Java runtime systems has significantly improved; hence, an even bigger performance difference could be expected if such a comparison was made at the time of writing.

3.2.4. Java Isolation API and MVM. The urgent need for component isolation within standard JVMs has been realized also by the industry. In the context of JSR-121 [20], a Java Isolation API has been defined. The core of this API is an abstraction called *Isolate*, which allows to strongly protect Java components from each other. The Isolation API ensures that there is no sharing between different Isolates. Even static variables and class locks of system classes are not shared between Isolates in order to prevent unwanted side effects. Isolates cannot directly communicate object references by calling methods in each other, but have to resort to special communication links which allow to pass objects by deep copy. An Isolate can be terminated in a safe way, releasing all its resources without affecting any other Isolate in the system.

The Isolation API follows a rather minimalistic approach in order to enable implementation across a wide variety of systems, including the Java 2 Micro Edition (for embedded devices). In particular, the communication model is very simple and may cause high overhead if large messages are transmitted frequently between different Isolates.

The Java Isolation API is supposed to be supported by future versions of the JDK. For the moment, it is necessary to resort to research JVMs that already provide the Isolation API, such as the MVM [11]. The MVM is a modified version of the Sun JDK 1.5.0 HotSpot Client VM supporting multiple Isolates within a single JVM process. Unfortunately, the MVM is currently only available on SPARC systems running the Solaris operating system. Moreover, as the more efficient HotSpot Server VM is not yet supported, there may be performance problems for complex applications.

4. Resource Management. In this section we explain why resource management, a missing feature in current standard JVMs, is needed in future Java runtime systems and we discuss several approaches to add resource awareness to the JVM.

4.1. Requirements. Using current standard Java runtime systems, it is not possible to monitor or limit the resource consumption of mobile agents. In order to ensure a secure, reliable, and scalable system that optimally exploits available resources, such as CPU and memory, support for resource management is a prerequiste. Resource management comprises resource accounting and resource control. Resource accounting is the non-intrusive monitoring of resource consumption, whereas resource control implies the enforcement of resource limits. Resource accounting helps to implement resource-aware agents that adapt their execution according to resource availability (self-tuning). Resource control is essential to guarantee security and reliability, as it allows to prevent malicious or erroneous agents from overusing resources. E.g., resource control is crucial to prevent denial-of-service attacks.

Below we summarize our requirements for the provision of resource management features in the JVM:

- Support for resource accounting as well as resource control.
- Support for physical resources (e.g., CPU, memory) as well as for 'logical' resources (e.g., number of threads).
- Extensibility—Support for user-defined, application-specific resources (e.g., database connections in an application server).
- Flexibility—Support for user-defined resource consumption policies.
- Low overhead.
- Compatibility—Existing agent code shall be executable without changes.
- Portability—Implementations shall be available on a large number of different systems.

4.2. Solutions. As before in Section 3, we distinguish between resource management libraries that are compatible with standard JVMs and specialized JVM implementations.

4.2.1. JRes. JRes [13] was the first resource management system for standard JVMs. It takes CPU, memory, and network resource consumption into account. The resource management model of JRes works at the level of individual Java threads. In other words, there is no notion of isolated component, and the implementation of resource control policies is therefore cumbersome. JRes is a pure resource management system and does not enforce any protection of components. However, JRes was integrated into the J-Kernel [29] (see Section 3.2.1) to support isolation and resource management within the same system.

JRes relies on a combination of bytecode instrumentation and native code libraries. To perform CPU accounting, the approach of JRes is to make calls to the underlying operating system, which requires native code to be accessed. A polling thread regularly obtains information concerning CPU consumption from the operating system. For memory accounting, JRes uses bytecode rewriting, but still needs the support of a native method to account for memory occupied by array objects. Finally, to achieve accounting of network bandwidth, the authors of JRes also resort to native code, since they swapped the standard `java.net` package with their own version of it.

The drawbacks of JRes are the limited number of supported resources and the lack of extensibility. Moreover, as JRes depends on native code and on specific operating system features, it is not easily portable. Another problem is the use of a polling thread to obtain CPU consumption information, as the regular scheduling of this thread is not guaranteed (see Section 2.2.3). Hence, the activation of the polling thread is platform-dependent and an eventual CPU overuse may be detected too late.

4.2.2. J-SEAL2. The J-SEAL2 system [3] also supports resource management [7]. In contrast to JRes, J-SEAL2 is implemented in pure Java, i. e., it is fully portable and compatible with any standard JVM. J-SEAL2 makes heavy use of bytecode instrumentation for CPU and memory management. Like in the case of JRes, the set of supported resources is rather restricted: In the default configuration, only CPU, memory, the number of threads, and the number of isolated components can be managed.

J-SEAL2 focuses on CPU management [7]. In order to achieve full portability, J-SEAL2 does not rely on the CPU time as metric, but it exploits the number of executed JVM bytecode instructions as platform-independent, dynamic metric [15]. In the J-SEAL2 system, each thread keeps track of the number of bytecode instructions it has executed within a thread-local counter. Periodically, a high-priority supervisor thread executes and aggregates the CPU consumption (i. e., the number of executed bytecodes) for all threads within each isolated component. This supervisor thread may also take actions against an overusing component: It may terminate the component or lower the priority of the component's threads. As in the case of JRes, the use of a high-priority thread to check CPU consumption is flawed by the underspecified scheduling semantics of Java threads.

While the use of the bytecode metric for measuring CPU consumption has many advantages [7], the J-SEAL2 approach also suffers from several drawbacks: The execution of native code cannot be accounted for (including also garbage collection and dynamic compilation), and the relationship between CPU time and the number of executed bytecode instructions remains unclear. The complexity of various bytecode instructions is very different. Furthermore, in the presence of just-in-time compilation, the execution time for the same bytecode instruction may depend very much on the context where the instruction occurs. Nonetheless, at least for simple JVM implementations, it may be feasible to assign weights to (sequences of) bytecodes in order to estimate CPU time on a particular system [26].

4.2.3. J-RAF2. J-RAF2[5], the Java Resource Accounting Framework Second Edition [5, 19], builds on the resource management ideas that were first integrated into J-SEAL2 [7]. In contrast to J-SEAL2, J-RAF2 does not assume a particular component model, it is a general-purpose resource management library, similar to JRes. However, in contrast to JRes, J-RAF2 is implemented in pure Java to guarantee portability.

The most important improvement of J-RAF2 over J-SEAL2 concerns CPU management. J-RAF2 does not rely on a dedicated supervisor thread to enforce CPU consumption policies. J-RAF2 uses an approach called ‘self-management’, where each thread in the system periodically invokes a so-called CPU manager that enforces a user-defined CPU consumption policy on the calling thread. The regular invocation of the CPU manager is achieved by bytecode instrumentation, where polling code is inserted in such a way that the number of executed bytecode instructions between consecutive polling sites is limited. I.e., polling code is injected before and after method invocations (call/return polling [16]) and in loops. This approach has the big advantage of not depending on the scheduling of the JVM. Thanks to several optimizations, the average overhead of CPU management could be kept reasonable, about 15–30% depending on the JVM [6].

4.2.4. NOMADS and the Aroma VM. NOMADS [25] is a mobile agent system which has the ability to control resources used by agents, including protection against denial-of-service attacks. The NOMADS execution environment is based on a Java compatible VM, the Aroma VM, a copy of which is instantiated for each agent. Resources are managed manually, on a per-agent basis or using a non-hierarchical notion of group. The major limitation of the Aroma VM is the lack of a just-in-time compiler, resulting in low performance.

4.2.5. Resource Management API. The work by Czajkowski et al. [12] is the first attempt to define a general-purpose, flexible, and extensible resource management API for Java, which has been validated in the MVM prototype [11].

The API supports user-defined resources, which are specified through a set of attributes that define the resource semantics. The API allows for resource reservations and notifications that are issued upon user-defined resource consumption situations. Such notifications ease the implementation of resource-aware program behaviour. Moreover, notifications can be synchronous and prevent actions that would exceed a given resource limit from succeeding, which is essential for resource control.

One limitation of the resource management API is that it requires a JVM with support for Isolates [20], because the Isolate is the smallest execution unit to which resource management policies can be applied. On the one hand, this approach simplifies the management of resources that can be handed over from one thread to another one, such as allocated heap memory (objects). As threads are confined to a single Isolate throughout their lifespan and object references cannot be shared across Isolate boundaries, handover of allocated heap memory across Isolate boundaries is not possible. Hence, managing such resources at the level of Isolates avoids complications due to the change of resource ownership. On the other hand, there are resources for which management at the level of threads can be very useful, such as CPU time. For instance, assume a service component that manages a pool of threads to handle incoming requests. A CPU management policy may want to limit the CPU time spent on processing a particular request, which would require binding individual threads to the CPU management policy. Unfortunately, the resource management API presented in reference [12] is not well adapted for such a scenario.

The need for a standardized resource management API has been realized also by the industry. In the context of JSR-284 [21], an improved resource management API is now under development.

5. Conclusion. The widespread distribution of Java as well as the mass of code and support that is available to Java developers makes it a sine qua non for mobile agent systems. This said, Java is probably the best and the worst that happened to mobile agents. The best because developing and deploying mobile agent systems became easy and highly portable; the worst because it is next to impossible to preserve Java’s usefulness and to build a sufficiently secure system at the same time. In order to deploy industrial strength mobile agent systems that are robust against various forms of DoS as well as breaches of confidentiality, Java has to evolve from an application-level runtime system into a true operating

[5]`http://www.jraf2.org/`

system with proper accounting and application separation capabilities. In the Java Community Process, this path has already been taken.

REFERENCES

[1] G. Back, W. Hsieh, and J. Lepreau, *Processes in KaffeOS: Isolation, resource management, and sharing in Java,* In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000), San Diego, CA, USA, October 2000.

[2] G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau, *Techniques for the design of Java operating systems,* In: Proceedings of the 2000 USENIX Annual Technical Conference, pages 197–210, San Diego, CA, June 2000.

[3] Walter Binder, *Design and implementation of the J-SEAL2 mobile agent kernel,* In: The 2001 Symposium on Applications and the Internet (SAINT-2001), pages 35–42, San Diego, CA, USA, January 2001.

[4] Walter Binder, *Secure and reliable Java-based middleware—Challenges and solutions,* In: First International Conference on Availability, Reliability and Security (ARES-2006), pages 662–669, Vienna, Austria, April 2006. IEEE Computer Society.

[5] Walter Binder and Jarle Hulaas, *A portable CPU-management framework for Java,* IEEE Internet Computing, 8(5):74–83, Sep./Oct. 2004.

[6] Walter Binder and Jarle Hulaas, *Java bytecode transformations for efficient, portable CPU accounting,* In: First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005), volume 141 of ENTCS (Electronic Notes in Theoretical Computer Science), pages 53–73, Edinburgh, Scotland, April 2005.

[7] Walter Binder, Jarle G. Hulaas, and Alex Villazón, *Portable resource control in Java,* ACM SIGPLAN Notices, 36(11):139–155, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[8] Walter Binder and Volker Roth, *Secure mobile agent systems using Java: Where are we heading?* In: Seventeenth ACM Symposium on Applied Computing (SAC-2002), Madrid, Spain, March 2002.

[9] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java.* Addison-Wesley, Reading, MA, USA, 2000.

[10] Ciarán Bryce and Jan Vitek, *The JavaSeal mobile agent kernel,* In: First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99), Palm Springs, CA, USA, October 1999.

[11] Grzegorz Czajkowski and Laurent Daynès, *Multitasking without compromise: A virtual machine evolution,* In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), pages 125–138, Tampa Bay, Florida, October 2001.

[12] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce, *A resource management interface for the Java platform,* Software Practice and Experience, 35(2):123–157, November 2004.

[13] Grzegorz Czajkowski and Thorsten von Eicken, *JRes: A resource accounting interface for Java,* In: Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98), volume 33, 10 of *ACM SIGPLAN Notices*, New York, USA, October 1998.

[14] S. Doyon and M. Debbabi, *On object initialization in the Java bytecode,* Computer Communications, 23(17):1594–1605, November 2000.

[15] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge, *Dynamic metrics for Java,* ACM SIGPLAN Notices, 38(11):149–168, November 2003.

[16] Marc Feeley, *Polling efficiently on stock hardware,* In: The 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, pages 179–187, June 1993.

[17] Michael Franz and Thomas Kistler, *Slim binaries,* Communications of the ACM, 40(12):87–94, December 1997.

[18] James Gosling, Bill Joy, and Guy L. Steele, *The Java Language Specification.* Addison-Wesley, Reading, MA, USA, 1st edition, 1996.

[19] Jarle Hulaas and Walter Binder, *Program transformations for portable CPU accounting and control in Java,* Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation), pages 169–177, Verona, Italy, August 24–25 2004.

[20] Java Community Process. *JSR 121 – Application Isolation API Specification,* `http://jcp.org/jsr/detail/121.jsp`

[21] Java Community Process. *JSR 284 – Resource Consumption Management API,* `http://jcp.org/jsr/detail/284.jsp`

[22] B. W. Lampson, *A note on the confinement problem,* Communications of the ACM, 10:613–615, October 1973.

[23] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, *Overview of the IBM Java Just-in-Time compiler, IBM Systems Journal*, 39(1), 2000.

[25] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, Timothy S. Mitrovich, Brian R. Pouliot, and David S. Smith, *NOMADS: toward a strong and safe mobile agent system,* In: Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00), NY, June 2000.

[26] James David Turner, *A Dynamic Prediction and Monitoring Framework for Distributed Applications.* Phd thesis, Department of Computer Science, University of Warwick, UK, May 2003.

[27] Jan Vitek, Ciarán Bryce, and Walter Binder, *Designing JavaSeal or how to make Java safe for agents,* Technical report, University of Geneva, July 1998. `http://cui.unige.ch/OSG/publications/OO-articles/TechnicalReports/98/javaSeal.pdf`

[28] Jan Vitek and Giuseppe Castagna, *Seal: A framework for secure mobile computations,* In: Internet Programming Languages, 1999.

[29] T. Von Eicken, C.-C. Chang, G. Czajkowski, and C. Hawblitzel, *J-Kernel: A capability-based operating system for Java, Lecture Notes in Computer Science*, 1603:369–394, 1999.

[30] T. Wilkinson, *Kaffe—a Java virtual machine,* Web pages at `http://www.kaffe.org/`.

[31] F. Yellin, *Low level security in Java,* In: Fourth International Conference on the World-Wide Web, MIT, Boston, USA, December 1995.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 10, 2006