Multiprocessing and Portability for PDAs

Grzegorz Czajkowski

Sun Microsystems Laboratories
901 San Antonio Rd., MS MTV29-112, Mountain View, CA 94303
Grzegorz.Czajkowski@sun.com

Abstract

The role of small devices in the emerging all-connected computer infrastructure is growing. So are the requirements that the application execution environments face. Portability, mobility, resource scarcity, and security concerns aggravated by often unknown sources of executed code combine together to create a challenging design and implementation task. It has been extensively argued and demonstrated that safe languages can solve some of these problems. In this paper, we focus on the multiprocessing aspect. We argue, in the context of the Java™ programming language, that multiprocessing execution environments based on safe languages for small devices can be built. However, in order to achieve lightweight and robust designs one has to consider a departure from replicating time-proven, traditional OS structure in the Java Virtual Machine (JVM™).

1 Introduction

Several general features of computing on modern small devices and the expectations made with respect to roles these devices should play in a larger infrastructure (agent systems, ubiquitous computing, etc) make designing application execution environments challenging. Mobility magnifies problems of application protection because of the increased exposure to potentially malicious code. Resource control must be adequate to fend off wide classes of denial-of-service attacks but at the same time flexible enough to accommodate for rapidly changing workloads. Resources are typically much scarcer on PDAs than on "standard" computers, which makes the designer's life more difficult in yet another dimension. On top of that, heterogeneity coupled with the need for code mobility and cross-platform interoperability further increases the challenge.

Safe languages offer the promise of a safe multiprocessing execution environment for portable code. The promise is not realized yet, however. In the workstation and desktop world, one can always fall back (in an arguably non-scalable fashion) on the protection offered by an underlying operating system. Thus, more often than not, multiple applications written in the Java programming language [AG98,LY99] and running on the same machine are executed in *separate* JVMs (separate processes). In the small devices world often no such easy exit exists, since there may be no sufficient OS process model.

Despite a rather pressing need and motivation, there is no standard for running multiple applications in the same JVM. Existing solutions, based on class loaders [LB98], are not safe. They are also not general, because of the emergence of JVMs for small devices, which do not have to support class loaders [Jav99].

This short paper discusses application isolation and resource control from the perspective of a designer of an execution environment based on a safe language. The focus is PDAs where often little can be assumed about available hardware protection, OS assistance, or plentiful computational resources. The context of the discussion is design decisions made during an ongoing project to turn the KVM [Jav99] into a multiprocessing JVM suitable for the Palm [Palm99].

2 Isolation

Currently, executing multiple applications in the same instance of the JVM poses problems. Applications are not isolated at the level of data access, because static fields of classes are accessible to all classes loaded by the same class loader. Using multiple class loaders alleviates the problem, since each application can have its own classes loaded by its own class loader. The solution is not complete, since system classes are still shared. This approach may also waste resources, since some classes may have to be parsed and JITed many times when used by many applications.

Running a separate JVM/process for each application does not scale up. It also does not scale down to small devices where the OS may not allow running of more than one process. The common wisdom defense of this approach is on the grounds of reliability and follows more or less along the lines of *"no JVM is as reliable as any commercially available OS"*. This is certainly true now but does not have to be in the future – after all, an OS kernel is also a piece of software. What distinguishes it from other pieces of software (apart from its functionality) is tremendous debugging effort, far from anything any JVM implementation has ever been subject to. Ultimately, one trusts software, whether it is an OS or a runtime of a safe language. The reliability issues of the Java platform and of an OS kernel are essentially the same. Moreover, a program coded in a safe language has less potential to crash due to software problems.

2.1 Data protection

Consider straightforward multiprocessing in the Java programming language: all applications share all classes. Since it is a safe language, there is already some built-in support for isolating applications from one another: data references cannot be forged. The only data exchange mechanism (barring explicit inter-application communication) is through static fields. In the absence of application-defined native code, this can be done only by explicit manipulation of static fields or by invoking methods that access these fields. This can lead to unexpected and incorrect behavior depending on how many applications use the same class with static fields.

The above observation suggests an approach for achieving separation among applications: maintain a separate copy of static fields for each class, one such copy per application that uses a given class. However, only one copy of any class should exist in the system, regardless of how many applications use it, since methods cannot transfer data from one application to another after the static fields communication channel is removed. (Removing covert communication channels, apparently impossible in general [Lamp73], is beyond the scope of this paper). Static fields for a new application are properly initialized (modified static initializers are invoked whenever necessary.) Figure 1 contrasts the sharing and isolation in (i) the most straightforward (and too simplistic) approach, (ii) the approach based on class loaders, and (iii) the new model.

In the absence of VM-specific features (Sec. 2.4) this idea combines the best isolation features of the OS based approach with the scalability of a single JVM approach. Moreover, API is unchanged. In particular, existing bytecode does not have to be modified to execute under the proposed isolation model.

A deficiency of our approach when compared to class loaders is that only one version of any given class can be loaded into the system. Thus, dealing with changing class implementations and dynamically loading new versions of them for new instances of applications is impossible in our current design.

In certain cases, the virtualizing of static fields described above may have to be turned off. For instance, consider `System.out` or its equivalent. It is important to ensure that each application has an access to `System.out` (if the security policy of a given environment allows this) and, at the same time, this static field is not directly shared by the applications. System properties are another example. In general, resources that must be shared by all classes have to be identified for each particular implementation of the JVM. Such cases are rare, though. Manually dealing with them may be necessary for a handful of system classes only.

2.2 Code access

Sharing classes among multiple applications may lead to the following problem concerning static synchronized methods. A thread may enter such a method and then immediately it may be suspended by another thread from the same application. This is a serious denial-of-service problem since the suspended thread still holds a monitor and no other application is able to call the method. Thus, in the new isolation model, class monitors must be virtualized so that there is a class monitor for each application. The objective is to ensure that proper mutual exclusion takes place among the threads of each application but one application cannot prevent anyone else from using a given synchronized method.

2.3 Communication

The proposed isolation model favors communicating via data copying (i.e. no sharing). The main reasons are the ease of resource management (no sharing facilitates termination and does cause problems with resource accounting) and easily controllable data separation. However, it can be argued that for some classes of applications data sharing is more appropriate.

2.4 VM-specific issues

The separation discussed above is sufficient to turn an idealistic implementation of the JVM into a multiprocessing environment with strong application isolation properties. However, most JVMs have implementation-specific features that improve performance but at the same time make the isolation more difficult to achieve.

Native code and thread termination are such issues. In our KVM implementation, they were addressed relatively easily, mostly because of a simple user-level thread model. Other facts we took advantage of are that (i) there is no user-defined native code, and that (ii) system native code cannot be entered simultaneously by more than one thread, is non-blocking, and its execution cannot be interrupted by a context switch. Isolating the applications from one another and the ability to terminate threads allows for clean application termination. VM-specific issues must be addressed in

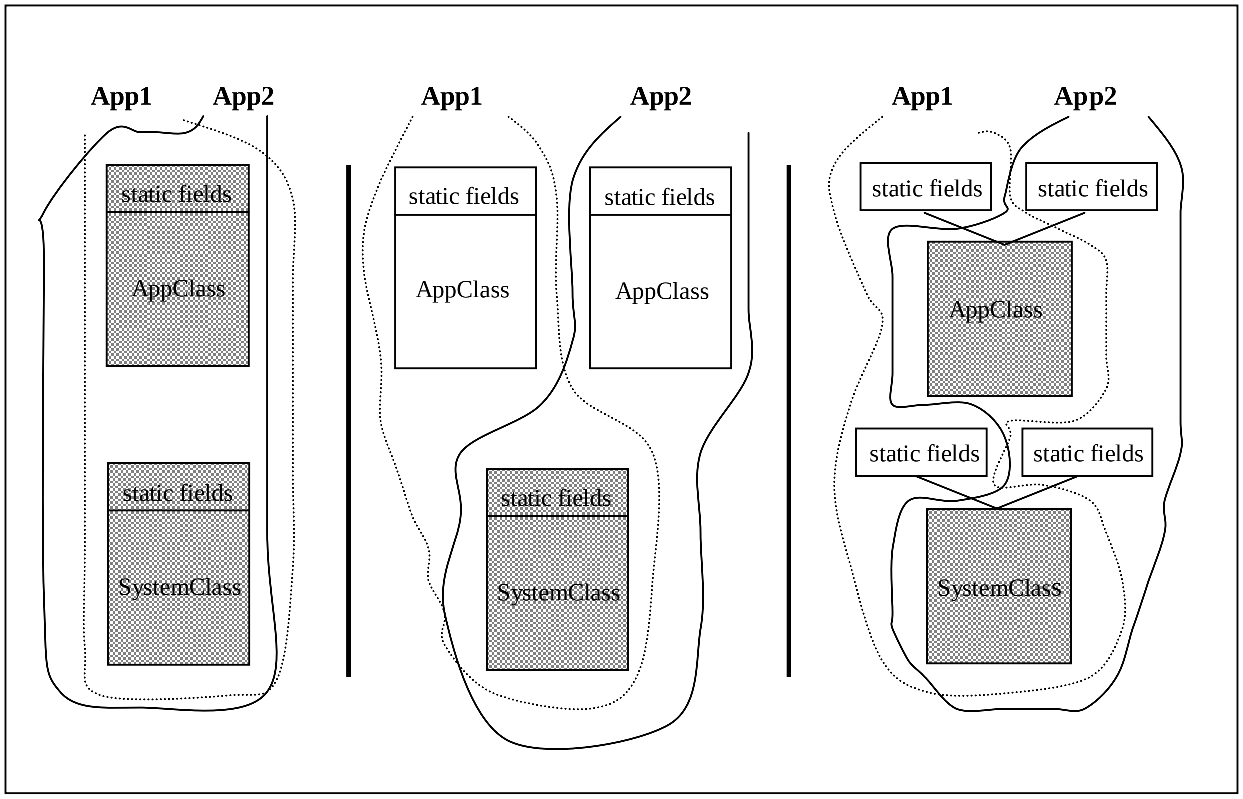

Figure 1. The naïve approach to multiprocessing ("share everything") in the Java programming language is shown at the left; the center shows the class loaders based solution; the right presents the new model. Shaded parts are shared by all applications; all other pieces are owned exclusively by a single application.

their specific contexts in order to turn the presented approach into a fully functional system.

3 Resource Control

Controlling resources consumed by programs written in the Java programming language is the topic of several recent publications [BTS+98,CvE98,HCC+98]. The focus has been on particular issues - how to account for resources, and how to make sure that resource limits are not exceeded, how to isolate resource consumption of one applications from this of other ones. The projects work with a mental model of a typical operating system and its fixed set of computational resources to manage (CPU time, memory, network, disk I/Os). The hardware protection/OS mindset manifests itself in the avoidance of resource sharing. For instance, Alta/GVM allocate physically separate heaps for different applications executing in the same JVM [BTS+98]. Manageability concerns justify such approaches but designs promoting sharing are more attractive when resources are scarce.

3.1 Shared heap

Separate per-process heaps are not adequate for environments with small amounts of memory (like, for instance, PalmV Organizer, with only 96K of RAM). Separate heaps do not allow best-effort memory allocation. Memory is pre-allocated for an application and can be underutilized. These facts made us choose a single, shared heap.

Due to our separation scheme, which virtualizes static fields, applications cannot share data. Thus, their garbage collection root sets are also disjoint. This means that the heap is logically partitioned into a set of separate sub-heaps and garbage collection can be invoked concurrently; moreover, it is easy to determine how much memory any given application is using.

This scheme enables dynamic changes to how much memory an application can use. While it is easy to give an application more memory, having applications release memory (for instance, when a new application has to be launched) is more problematic. In our design, applications are given a guaranteed amount of memory; anything beyond that is granted only if memory is available but the application must be prepared to release it. If an application wants to use more memory than its guaranteed amount, it registers a callback to be notified whenever the surplus memory has to be released. The practical appeal of this approach remains to be seen.

3.2 Growing set of resources

A system written entirely in a safe language, without any support from an operating system/hardware protection poses additional challenges with respect to resource control. Different components of the system are shared differently than in an operating system. For instance, consider a garbage collector (which does not even appear in a traditional OS but is an integral part of a safe language) and consider in how many ways a malicious/buggy application can attempt to abuse it.

The most obvious attack is to try acquiring so much memory that other applications cannot get any of it. This problem is well understood, one possible solution is to monitor the amount of memory allocated on a per-application basis and reject requests for more. Another attack is to allocate objects at a fast rate but discard them right after the allocation. This stresses the garbage collector (GC). A solution is to add resources consumed by the GC to the accounting information pertaining to a given application. Yet another, admittedly quite sophisticated attack, aimed at non-compacting GCs, is to allocate and discard objects in such a pattern so as to lead to significant fragmentation. A solution here would be to try to allocate application's objects in larger contiguous chunks or to use separate heaps.

The preceding paragraph is structured as a sequence of "a possible attack is... a solution is...". However, this was just a GC example. We do not claim to know all sorts of (DoS) attacks one can fancy against a GC. Neither do we know all possible attacks against all resources (we cannot even say we know all resources that may be attacked in one way or another.) In our opinion this justifies the following design decision: a modular resource control system, where all sorts of resources (such as various measures of heap usage or screen real estate) can be registered and centrally monitored.

Why centrally? This provides a convenient place for enforcing various resource usage policies. For instance, resource tradeoffs [CCH+98] can be easily made in such a central place. In contrast, scattered managers of a single resource with different APIs (or sometimes without any external API at all) make it difficult to combine knowledge about consumption of various resources. Software engineering argument also favors modularity – it is easier to add new resources this way, since adding a new resource amounts to implementing a mechanism to gather information about its consumption; policies can be implemented separately. Figure 2 contrasts our approach with a more typical resource management structure.

4 Related Work

Space limitations allow us to only gloss over a few of the related projects. The most relevant one (and mentioned throughout the paper) is being carried out at the University of Utah [BTS+98]. Two operating systems have been designed that demonstrate how a process model can be implemented in the JVM. The first system, GVM, is structured much like a monolithic kernel and focuses on complete resource isolation between processes and on comprehensive control over resources. A GVM process consists of a class loader-based name space, a heap, and a set of threads in that heap. In addition to their own heaps (each process has its own heap) all processes have access to a special, shared system heap. For every heap, GVM tracks all references leading to other heaps and all references pointing into it.

Alta closely models a micro-kernel model with nested processes, in which a parent process can manage all resources available to child processes. Memory management is supported explicitly, through a simple allocator-pays scheme. The garbage collector credits the owning process when an object is eventually reclaimed. Because Alta allows cross-process references, any existing objects are logically added into the parent memory. This makes the parent process responsible for making sure that cross-process

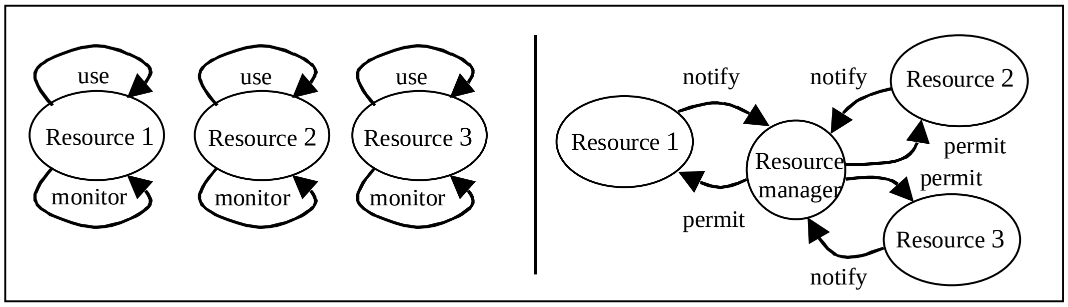

Figure 3. Central resource manager (right) vs resources being managed separately (left).

references are not created if full memory reclamation is necessary upon process termination. Both GVM and Alta are implemented as considerable modifications to the JVM. Both systems support strong process models: each can limit the resource consumption of processes, but still permit processes to share data directly.

An example of a class loader based approach to application protection is the J-Kernel [HCC+98]. The J-Kernel adds *protection domains* to the JVM, and makes a strong distinction between objects that can be shared between tasks, and objects that are confined to a single task. Each domain has its own class loader. The system, written as a portable Java library, provides mechanisms for clean domain termination (e.g. no memory allocated by the task is "left over" after it is terminated) and inter-application communication (performed either via deep object copy of method arguments and return values or via controlled sharing enabled by revocable capabilities).

Balfanz and Gong designed a multiprocessing JVM in order to explore the use of the Java security architecture to protect applications from each other [BG97]. The proposed extensions, based on class loaders, enhance the standard JVM so that it can support multiprocessing. An important part of the work is the clear identification of several areas of the JDK that assume a single-application model.

5 Conclusions

The approach to multiprocessing in Java outlined in this paper is, in our opinion, well suited for simple JVMs designed for PDAs. Isolating applications without having to rely on OS processes or class loaders and modular resource management make the design both lightweight and relatively easy to extend whenever a new DoS attack is found to be possible. In the absence of any inter-application communication mechanisms, the isolation model allows for clean application termination and exact resource accounting. This simplicity of the model made us opt for copy-only communication. Assessing the value of controlled sharing based on capabilities like in the J-Kernel or on a special heap like in GVM still needs to be investigated in these settings.

6 References

[BTS+98] Back, G, Tullmann, P, Stoller, L, Hsieh, W, and Lepreau, J. *Java Operating Systems: Design and Implementation.* TR UUCS-98-015, Department of Computer Science, University of Utah, August 1998.

[BG97] Balfanz, D., and Gong, L. *Experience with Secure Multi-Processing in Java.* TR 560-97, Dept. of Computer Science, Princeton University, September 1997.

[CvE98] Czajkowski, G., and von Eicken, T. *JRes: A Resource Control Interface for Java.* ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

[CCH+98] Czajkowski, G., Chang, C-C., Hawblitzel, C., Hu, D., and von Eicken, T. *Resource Management for Extensible Internet Servers.* 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998

[GS98] Gong, L. and Schemers, R. *Implementing Protection Domains in the Java Development Kit 1.2.* Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.

[HCC+98] Hawblitzel, C., Chang, C-C., Czajkowski, G., Hu, D. and von Eicken, T. Implementing Multiple Protection Domains in Java. USENIX Annual Conference, New Orleans, LA, June 1998.

[Jav99] Sun Microsystems, Inc. *The K Virtual Machine – A White Paper.* http://java.sun.com/products/kvm/wp.

[Lamp73] Lampson, B. *A Note on the Confinement Problem.* Communications of the ACM, 16(1), October 1973.

[LB98] Liang S., and Bracha, G. *Dynamic Class Loading in the Java Virtual Machine.* In Proceedings of ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

[LY99] Lindholm, T., and Yellin, F.. *The Java Virtual Machine Specification*. 2nd Ed. Addison-Wesley, 1999.

[Palm99] Palm Computing, Inc. *The Palm Computing Platform Development Zone.* www.palm.com/devzone.