

Type of submission: Technical session(s)

Keynotes session(s)

Topic covered from the Call for Papers:

Multi-application and open OS integration

Next-Generation Embedded Java Operating System for Smart Cards

Laurent Lagosanto

Gemplus Software Research Labs,
BP 100, 13881 Gemenos cedex, France
Phone: +33 (0)4 4236 4196, Fax: +33 (0)4 4236 5555
lago@research.gemplus.com

Abstract. This paper deals with both the development and production environments needed for future smart cards. Both are considered with a software-engineering standpoint. This approach results in an embedded Java operating system dedicated to support micro-server operations thanks to powerful Java and operating system features described in the first part of this paper. The second part of this paper describes the Java embedded operating system architecture and the production tools that enable a migration path from a development platform up to a mass-produced and customized platform.

Introduction

When the first Java Card specifications were issued, the card chips were roughly 8051-like derivatives (8-bits) with a few hundred of bytes of RAM and a few kilo bytes of EEPROM. These server constraints led the Java Card Forum and Sun to issue a specification [1] that had a lot of limitations compared to standard Java. This specification does not specify an operating system but an execution environment that every player in the smart card industry is able to implement on top of his own proprietary operating system.

The silicon market evolution seems to indicate that next generation products will be based on much more powerful machines (32bit, RISC, Cache are becoming common in chip catalogs). The smart card industry, through the Java Card Forum, has clearly identified this evolution and has started initiatives like the Java Card 3.0 specification process that aim to define the next generation version of Java Card adapted to next generation hardware.

Concurrently with this hardware evolution, Java has also evolved a lot. It is no more limited to the desktop environment and is now present in both the server market (with J2EE) and in the embedded market (with J2ME). This evolution brought interesting features to the Java platform: techniques for running long-lived applications are common on the server-side, and the minimal Java subset applicable to categories of devices has been defined with the J2ME configurations [2].

The purpose of this paper is to have an overview of what can be considered now as the minimal Java features that can be integrated to modern smart card hardware and how these features can be turned into a card Operating System that cope with the smart card industry particular constraints like mass-production, personalization, post-issuance. The reason for targeting an operating system instead of an additional software layer like it's done in the PC world with standard JDK is to use Java as *the unique* hardware abstraction for applications without intermediate levels, to reduce the footprint and maximize efficiency.

The rest of this paper is organized as follows: the first section describes the desirable features for a Java OS and the second section explains how they are engineered in different platform configurations.

Java Operating System Features

The Java platform is defined in two separate specifications: the Java Virtual Machine Specification [3] and the Java Language Specification [4]. In the context of operating system these specifications are only relevant at the execution environment level, thus the selection of the Java OS features is guided by the classical needs of an operating system: code loading/linking, execution engine and memory management.

The code delivery format: the Java Class file

The class file is defined in the JVM spec [3], as the delivery format for the code of classes to load into a Java VM. This format is considered to be verbose and not compact, but it has some other properties that provide a maximal flexibility, both for development, deployment, and application code update. Other formats such as the Java Card 2.x CAP file or the JEFF file format [5] are focusing on file size reduction using techniques like linking information removal or file structure reorganization. These techniques have good result in size reduction but induce deployment/development constraints depending on the degree of size reduction. For example a CAP file can be 10% the size of the equivalent set of class files, but it requires an off-card conversion step for the developer and to maintain off-card a database of linking information to perform off-card pre-linking, which is a painful task for deployment and card management..

We argue that on-card class file processing that provides the same reduction ratio as the JEFF conversion (50% reduction). The loading process can then be similar to the following figure:

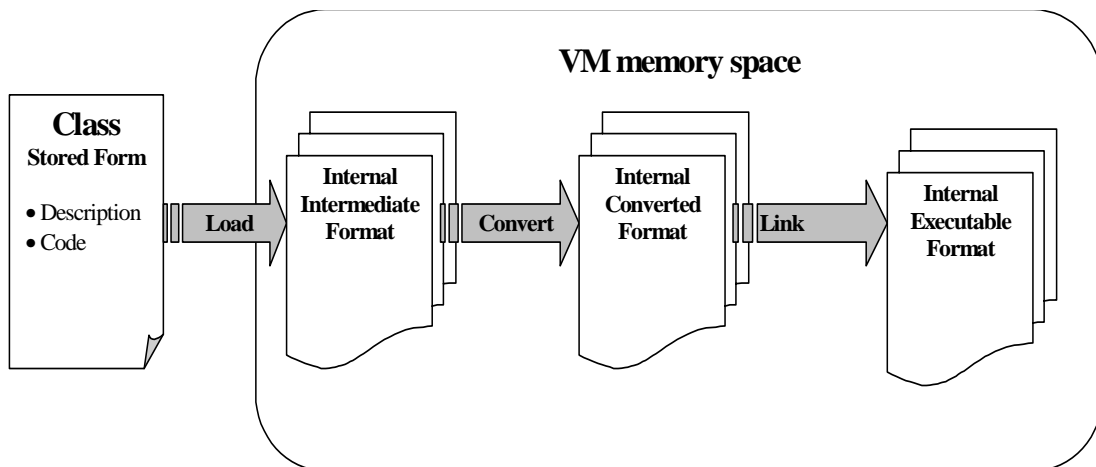


Figure 1: code loading/linking as a pipelined process

Since the load/link operation occurs only once in the life-time on the class, we think that the development/deployment/interoperability benefits brought by loading standard class files are worth the effort of loading the class file in its entirety and applying an on-board conversion process that enables to store in card memory a structure equivalent to 50% of the original class file size.

One of the main industrial advantage of choosing the class file and defining your own conversion process is that it provides freedom for differentiation: every card manufacturer can choose an internal format adapted to his own particular constraints.

The Multithreaded Execution Engine

The Java platform has native support for multithreading and synchronization, and even in the lowest configurations defined for the embedded market like CLDC [6], this support is maintained. One may think that for embedded and resource constrained devices the support for multithreading may be too heavy to be reasonable. Actually, there are several reasons to keep multithreading support in the core of a very small Java operating system. The main reason is that it enables to write powerful application frameworks on top of the OS, in Java, without needing complex native modifications of the OS like it has happen when the SIMToolkit application model [7] has been introduced in Java Card. There are also obvious “classical” reasons to use threads that are extensively described in the literature [8]

In such memory-limited devices as smart cards, the VM implementers must carefully choose an efficient and resource-saving implementation of the multithreading support. The existence of Java application only on top of a Java operating system, collaborating with native code and not with native applications like in a desktop PC seems to favors a green-thread like implementation rather than a native-thread implementation.

Unified Memory Management: a Garbage Collector for Java *and* System Objects

One very popular feature of the JVM is the automatically garbage-collected heap that is used to allocate Java objects when they are created, letting the burden of compacting, moving, freeing the memory to the system instead of the developer.

This feature was missing in the Java Card specification and thus leads the developer of complex applications to develop their own memory management routines within the application code, written in Java. This is typically a failure of the Java Card to behave as an OS: the applications shouldn't handle such sensitive task. There are several types of Garbage Collectors but the ones that seem the more appropriate to smart card-like environments are the generational collectors. This type of collector are well adapted to server-like long-lived applications which is typically the case of smart card applications.

Traditionally, the memory management for object instances is separated from the memory management of system data like classes description, code, threads. This usually leads to complexity, bigger footprint of the memory management and static segmentation of the memory. We are experimenting an alternative to such a split organization that is to use a unified object-oriented memory manager that handles both applications and system objects. In this scheme, everything in the system is an object, including Java object instances, code, loaded classes, and every object in the system is subject to garbage collection.

This unified memory management leads to a simpler management code, which is then probably smaller and more robust.

Allowing Many Different Application Models

All the previous features of the JVM described above have a common objective: put in the OS the functionality that will make smaller and more efficient applications. One great strength of Java that was lost in Java Card and that we think must be reconsidered for next generation smart card OS is to be application model independent: the JVM doesn't specify a particular application model, but provides all the core features that are necessary to build various successful frameworks such as Applets, Servlets or RMI Objects.

Platform Configurations

The smart card application and system designers and developers have considerations that are very similar to the people working in the mass-production embedded devices market: how to debug quickly/safely the OS and the applications, how to build quickly a product configuration, and how to decline an OS in a product range to leverage the investment. Furthermore, the smart card industry has his own particularities like for instance the need for personalization of every issued card with the holder information.

Development Issues

The use of Java is in itself the beginning of a solution for some of these problems, because of the wide range of products freely or commercially available to help designers and developers. But the high safety exigency on smart-card products implies that a maximum effort must be made on any tools that would allow the developers to operate in an environment as close as possible to the actual device. Debugging the code while it runs

Production Issues

Other tools are needed in the production and personalization phase of the cards. The ROMizer intervenes at the production step, to generate the code to mask into the chip's ROM. It is complemented by the Memory Serializer which is used to build memory images stored in files, that will be sent to the card at initialization and personalization time to build the right memory structure that will put the card in the right state. Such tools are usually difficult to write and to maintain, but these tasks can be greatly simplified using the introspection capabilities provided by the unified object-oriented memory management introduced above.

A Configurable Integrated Operating System

A solution to minimize the cost of these tools is to build an operating system that includes them from the beginning, in a configurable manner, and that can be declined from the same set of sources in different editions adapted to the targeted device.

Such a configurable OS is illustrated in the following figure.

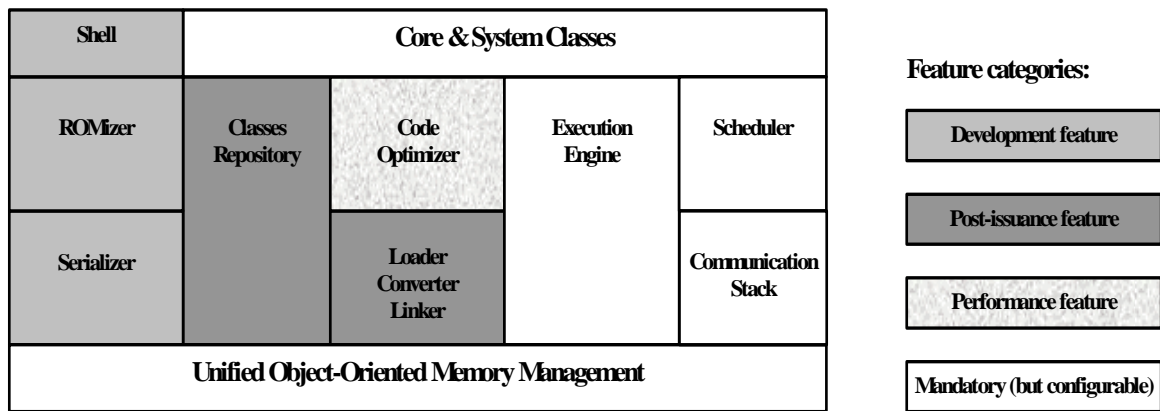


Figure 2: A configurable operating system

For example, a “developer edition” of the OS would include any developer features like an embedded debugging interface based on the JPDA standard [9], a shell-based console to administer the OS (load/remove classes, create threads,...). Such a complete edition would of course run on PC and on emulators only, not on real cards. Another declination would be the “post-issuance enabled edition” that would be the high-end embeddable version of the OS, targeted for high-end cards with post-issuance facilities (with a loader/linker). On the other extremity would be the “minimal edition”, targeted for low-end cards, without any post-issuance capabilities, that would have a much smaller footprint.

Conclusion

In this paper we have highlighted some features of an embedded Java operating system suitable for next-generation smart cards considered as Java micro-server platforms. Three important Java operating system characteristics have been described (class file format acquisition, multithreading support, and a unified memory management). Their implementation has been shown feasible, and their usage and benefits has been described in the context of a configurable card platform architecture. Thanks to tools such as shell console, debugger, ROMizer and serializer, the migration path from a rich platform configuration up to a customized platform configuration has been explained.

The overall benefit of this embedded Java operating system architecture is its adaptability for being used, on one hand as a full-fledge Java environment to rapidly develop and test applications, and on another hand as an engineering environment for mass-production of optimized embedded code hosted by limited devices. Though

originally developed in the context of smart cards, such an architecture could be suitable in any embedded device in which the Java development platform can differ from the final Java deployment platform.

References

- [1] The Java Card Virtual Machine specification 2.1.1. Sun Microsystems
<http://java.sun.com/products/javacard/>
- [2] Java 2 Micro Edition, Sun Microsystems
<http://java.sun.com/j2me/>
- [3] Java Virtual Machine Specification,
Tim Lindholm, Frank Yellin. Sun Microsystems.
<http://java.sun.com/docs/books/vmspec/index.html>
- [4] Java Language Specification,
James Gosling, Bill Joy, Guy Steele, Gilad Bracha. Sun Microsystems.
<http://java.sun.com/docs/books/jls/index.html>
- [5] The JEFF File Format, J-Consortium
<http://www.j-consortium.org/jeffwg/index.shtml>
- [5] J2ME Connected Limited Device Configuration
<http://java.sun.com/products/cldc/>
- [6] Sim API for Java Card, ETSI TS 143 019 V5.2.0 (2002-03)
<http://www.etsi.org>
- [7] *Concurrent Programming in Java*. D. Lea. Sun Microsystems, 1999.
ISBN 0-201-31009-0, Addison-Wesley
- [8] Java Platform Debugger Architecture, Sun Microsystems,
<http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html>