Volume 1,Issue-1,December-2011 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research paper

Available online at www.ijarcsse.com

Use of Applets for Java Cards Technology For Smart Cards

Ravi Kant Vyas
CSE, ITM Bhilwara
vasravikant@gmail.com

Anurag Sharma
ECE, ITM Bhilwara
sharanurag@gmail.com

Neha Palod
CSE, ITM Bhilwara
nehapalod10@gmail.com

***Abstract*— Java Card technology enables smart cards and other devices with very limited memory to run small applications, called applets, which employ Java technology with a secure and interoperable execution platform that can store and update multiple applications on a single device. This accelerated process reduces development costs, increases product differentiation, and enhances value to customers. In a manner complementary to the Standard, Enterprise, and Mobile editions of the Java 2 Platform, Java Card technology makes it easy to integrate security tokens into a complete Java software solution. A smart card runtime environment must provide the proper transaction support for the reliable update of data, especially on multi application cards like the Java Card. The JAVA CARD transaction mechanism allows to protect sensitive operations on smart cards against problems due to card tears or power losses. Statements within a transaction are viewed as a single atomic operation so that either they are all performed or none of them is. We consider security problems that can be caused by a card tear. One of the most serious problems of Smart Card System such as Java Card is about an amount of insufficient memory. Moreover, most of the embedded devices with Java VM support the post-issuance of new applications. These applications are usually stored in the memory, which is usually EEPROM.**

I. INTRODUCTION

Generally, there are 256 byte code instructions in Standard Java. However, Java Card is implemented as a subset of the Standard Java for a limitation of a typical resource-constrained device. We discover that instructions of 186 to 253 are neither specified nor reserved. These free instructions can be transformed into new instructions for improving the performance of time and space.

Smart card support is a component of the IT infrastructure in a growing number of sectors [5]: banking, mobile and non-mobile communications, ID/access, government, multimedia, etc. Most of the applications require a high-degree of reliability. Java Card [6] is one of the leading technologies in this sector as it provides significant features: multiple applications, portability, and compatibility with a popular programming language technology (Java).

The B method [7] is a good candidate for such process. Based on the experience gathered from the development of formal specification languages such as VDM and Z, B is one of the foremost formal methods with a strong industrial support. Smart cards provide the secured access to stored data. Data on the smart card is usually not accessible for an external application until it has authenticated itself to the card sufficiently. If the communication only consists of read accesses, the card can deliver the requested data without compromising the security and integrity of the stored data. If the external application creates or updates data on the card, care must be taken that the integrity of the data is preserved throughout the communication. Either all updates take place during the communication or the data on the card is reverted to its initial state in case of an interrupted execution.[2]

The terminal applications set up and control the communication with the smart card and mostly also control the consistency of their data on the card completely. Current applications typically flag their data on the card to be inconsistent with the first write access during a series of updates. After all updates, the terminal application finally records its data on the card to be consistent again. If a terminal application is confronted with a card in an inconsistent state, its state may be reset by the terminal application itself, but more often must be fulfilled under special authority within a trusted environment. The dependency of the smart card consistency on external applications can be accepted as long as the smart card is only used for a few critical applications where any irregularity must be recorded and checked at a central site. Otherwise, a smart card should not only be able to verify the access rights of an external application, but should also provide a tighter control over the consistency of the internally stored data. Especially, on multi application cards where each application on the card has access to its own data, applications must also be able to control the integrity of their data. Thus the underlying system must provide a proper transaction mechanism which ensures the correct transition between consistent states of applications and offers its functionality to all applications residing on the card. The task of the system is then twofold. First, the system is required to ensure that all updates of an application are performed atomically; second, it must perform crash recovery to provide stability: the system must recover its state and the state of the applications to a consistent state if a transactional computation fails. A simple transaction model on the card may only support user level transactions in the traditional sense. Transactions can be assumed to begin and end within the communication with a terminal application, are thus short lived and need not be split in multiple sub transactions even if multiple applications cooperate together. However, the implementation of a transaction mechanism is hindered by the extremely limited resources on a smart card. With RAM capacities around 1 KByte and writable EEPROM capacities around 16 KByte the

transaction implementation must be carefully chosen. In case of the Java Card, the underlying standard Java environment must first be extended to offer integrated transactional computations. The familiar programming convenience of Java should be retained while the necessary resource demands must be kept as minimal as possible. [2]

The JAVA CARD programming language [4] is a JAVA dialect tailored to the development of applications for smart cards, called applets. Its core language is a strict subset of the JAVA object-oriented language with a set of restrictions due to the resource-constrained smart card environment: there are neither floating-point numbers, strings of characters, threads, nor multi-dimensional arrays. However, because of the specific nature of smart cards it contains some additional features available through the system API [22]. Among them is a so-called transaction mechanism. It is justified by security issues concerning the possibility of tearing out a card from its reader at any moment during a session. In such a case the consistency of sensitive data, persistently stored on the card, should be preserved.[3]

II. JAVA CARD INTRODUCTION

Microsystems publish the Java Card Platform Specification and the Java Card Development Kit, which includes a reference implementation based on the specification. Providing the basis for cross-platform and cross-vendor applet interoperability, version 2.2.2 of the specification includes three documents:

The Java Card Virtual Machine Specification defines the features, services, and behavior that an implementation of the Java Card technology must support.

The Java Card Runtime Environment Specification defines the necessary behavior of the runtime environment (RE) in any implementation of the Java Card technology.

API for the Java Card Platform complements the Java Card Runtime Environment Specification, and describes the application programming interface of the Java Card technology.

The Java Card Development Kit is a suite of tools for designing implementations of the Java Card technology, and for developing applets based on the Java Card API Specification.

TABLE I
Java Card Component size in Kilobytes

Virtual machine	4.0
Memory management subsystem (including transaction support)	4.0
Garbage collector for RAM and EEPROM	1.5
DES implementation (no hardware)	1.4
RSA/DSA implementation (PK coprocessor)	2.4
On-card RSA/DSA private-key generation	0.6
PK hash algorithm (SHA1)	0.6
T = 1 protocol	1.0
T = 0 protocol	0.5
T = CL (contactless) protocol	0.5
Java Card system classes (no crypto)	2.5
Java Card crypto classes (IBM proposal)	0.7
Java Card crypto classes (JC21)	5.0
OP implementation (mixed native/Java code)	8.0
Full-fledged Security Domain support	1.0
Applets required to be in ROM for VOID compliance	1.2

A. *Applet Execution*

The Java Card environment shares the basic architecture with the standard Java environment. However, due to the limited resources on current smart cards the Java Card sacrifices a number of Java features.

The runtime environment initiates the applet installation by calling the install() method of its class instantiating an applet object and registering it at the runtime environment.

An external application can initiate a session with the installed applet by selecting it first at the runtime environment. The select command will be forwarded by the runtime to the applet's select () method, each following command will be forwarded to its process () method. The applet processes each command and returns from its invocation with a response for the terminal application. Thus the invocation of the applet is event driven until the remote application finishes the card session or selects a different applet where the current applet is notified by the invocation of it's deselect () method.

B. Memory Management

The applet instance and associated persistent objects of an application are placed in the non volatile storage on a card, usually EEPROM, provides similar read and write access as RAM does, but with the important difference that the number of physical writes is limited and writes to EPROM cells are typically more than thirty times slower than writes to RAM. Performance of writes can be increased on many current chips by initiating block writes instead of multiple single EEPROM writes where individual bytes are written in parallel to EEPROM. Neither single byte nor block writes are guaranteed to succeed in case of sudden power loss, the write operation can suddenly fail after an arbitrary number of bits have already been written. Thus the runtime environment can only rely on the outcome of a single flag write as the basic building block for transactions. Both RAM and EEPROM size is extremely limited on current smart card hardware, ranging typically up to 1 KByte for RAM and up to 16 KByte EEPROM for current Java Cards.

In contrast to EEPROM, RAM loses its value in case of a power loss. For repeated, performance and security sensitive computations, RAM must be usable by Java Card applications. For instance execution state, operand stack and local variables must be placed in RAM by the virtual machine. Other than that, the Java Card 2.1 specification allows applets to allocate array instances explicitly in RAM. Our model extends the Java Card specification by allowing any type of object to be placed both in EEPROM as well as in RAM. The system is described in detail in and especially allows the easy deployment of a RAM garbage collector.

Data located in RAM, i.e. execution state and transient objects, is not considered to be part of the persistent state and its manipulations are not recorded during the transaction due to a number of reasons, among which are performance penalty and security implications. Thus, only changes to the applet objects in EEPROM must be covered by the transactional mechanism.[2]

III. JAVA CARD SPECIFIC FEATURES

The Java Card runtime and virtual machine also support features that are specific to the Java Card platform:
Persistence: With Java Card, objects are by default stored in persistent memory (RAM is very scarce on smart cards, and it is only used for temporary or security-sensitive objects). The runtime environment as well as the bytecode has therefore been adapted to manage persistent objects.

Atomicity: As smart cards are externally powered and rely on persistent memory, persistent updates must be atomic. The individual write operations performed by individual bytecode instructions and API methods are therefore guaranteed atomic, and the Java Card Runtime includes a limited transaction mechanism.

Applet isolation: The Java Card firewall is a mechanism that isolates the different applets present on a card from each other. It also includes a sharing mechanism that allows an applet to explicitly make an object available to other applets.
The JAVA CARD programming language contains several highly specific features that come from the characteristics of the smart card embedded environment. Some of them are induced by the nature of the underlying hardware, others are given as API for the programmer to handle security needs. In this paper we consider three of them for which we provide the support in the KRAKATOA tool: the way memory is organized, the so-called card tear property and the atomic transaction mechanism. We also reason about non atomic method calls but for sake of clarity we will introduce them later in this paper.

In JAVA CARD, the memory is organized in a completely different way than in JAVA. A distinction is made between persistent objects (stored in EEPROM) and transient (or volatile) objects (stored in RAM). The values of persistent objects are available during the whole card life cycle, it is typically the case for applet objects and their fields. However, transient data are cleared after each session between the card and a terminal because they do not survive in the absence of power. Generally, in JAVA CARD programs some large arrays are allocated in transient memory in order to compute auxiliary calculations faster.

Smart cards can be teared out of the terminal reader at any moment during a session. As a result all transient objects are cleared and persistent ones are left in the state they were at the precise instant of the power loss. A major issue for the security of an embedded application is to preserve data integrity and to maintain a coherent memory state in case a card tear occurs.

Another topical problem is to ensure data consistency in case a card tear occurs during a sequence of several updates to persistent memory. To that aim the JAVA CARD API provides a so-called transaction mechanism — as those of database systems — that makes a set of statements as if it was a single atomic operation. All the updates are effectively done or none of them is.

The class javacard.framework.JCSystem contains the methods for using the transactions. Concretely, a transaction starts with a call to the method beginTransaction() and the changes made from that point only become effective after commitTransaction() is called. A transaction can be aborted either by the system (i.e. the JAVA CARD Run time Environment) if a card tear — or more generally any sudden power loss — or a memory buffer overflow occurs, or by the programmer thanks to the method abortTransaction(). In such a case, persistent objects are reset to their values in the state just before the transaction. Transient memory is not at all affected by transactions, thus any assignment to a volatile variable is done unconditionally.[3]

In JAVA CARD, the memory is organized in a completely different way than in JAVA. A distinction is made between persistent objects (stored in EEPROM) and transient (or volatile) objects (stored in RAM). The values of persistent objects are available during the whole card life cycle, it is typically the case for applet objects and their fields. However, transient data are cleared after each session between the card and a terminal because they do not survive in the absence of power. Generally, in JAVA CARD programs some large arrays are allocated in transient memory in order to compute auxiliary calculations faster.

Smart cards can be teared out of the terminal reader at any moment during a session. As a result all transient objects are cleared and persistent ones are left in the state they were at the precise instant of the power loss. A major issue for the security of an embedded application is to preserve data integrity and to maintain a coherent memory state in case a card tear occurs.

Another topical problem is to ensure data consistency in case a card tear occurs during a sequence of several updates to persistent memory. To that aim the JAVA CARD API provides a so-called transaction mechanism — as those of database systems — that makes a set of statements as if it was a single atomic operation. All the updates are effectively done or none of them is.

The class javacard.framework.JCSystem contains the methods for using the transactions. Concretely, a transaction starts with a call to the method beginTransaction() and the changes made from that point only become effective after commitTransaction() is called. A transaction can be aborted either by the system (i.e. the JAVA CARD Run time Environment) if a card tear — or more generally any sudden power loss — or a memory buffer overflow occurs, or by the programmer thanks to the method abortTransaction(). In such a case, persistent objects are reset to their values in the state just before the transaction. Transient memory is not at all affected by transactions, thus any assignment to a volatile variable is done unconditionally. [3]

IV. SMART CARDS AND JAVA CARD

An increasing number of IT applications require that users provide data via a portable device. Moreover, for security

reasons, it is desirable that such devices include at least simple processing capabilities. The smart card technology answers these needs and gains acceptance and popularity. Unique amongst the different smart card operating platforms, Java Card provides vendor inter-operability and has now reached a de facto standard status in this industry [1].

The Java Card platform provides a subset of the Java programming language. It allows memory-constrained devices, like smart cards, to run applications in a secure and interoperable way. Security is obtained through Java elements, like its secure execution environment, which controls, for instance, the level of access to all methods and attributes; and the applet separation by a resource named applet firewall. Inter-operability is the characteristic that allows the execution of a Java Card application in any smart card that follows the Java Card specifications, independently of hardware and software manufacturers, without or with few code modifications.

The use of this technology brings many improvements for the developer of smart card applications. The ease of programming in Java, that abstracts the low level details of the smart card system; and Java development tools (like IDEs, simulators and emulators) allow a rapid application build, test and installation cycle, reducing the time and the cost of software production. Moreover, other benefits are the possibility of multiple applications to coexist in a same card and the ample compatibility with smart card international standards, like ISO 7816.

An increasing number of IT applications require that users provide data via a portable device. Moreover, for security reasons, it is desirable that such devices include at least simple processing capabilities. The smart card technology answers these needs and gains acceptance and popularity. Unique amongst the different smart card operating platforms, Java Card provides vendor inter-operability and has now reached a de facto standard status in this industry [1].

The Java Card platform provides a subset of the Java programming language. It allows memory-constrained devices, like smart cards, to run applications in a secure and interoperable way. Security is obtained through Java elements, like its secure execution environment, which controls, for instance, the level of access to all methods and attributes; and the applet separation by a resource named applet firewall. Inter-operability is the characteristic that allows the execution of a Java Card application in any smart card that follows the Java Card specifications, independently of hardware and software manufacturers, without or with few code modifications.

The use of this technology brings many improvements for the developer of smart card applications. The ease of programming in Java, that abstracts the low level details of the smart card system; and Java development tools (like IDEs, simulators and emulators) allow a rapid application build, test and installation cycle, reducing the time and the cost of software production. Moreover, other benefits are the possibility of multiple applications to coexist in a same card and the ample compatibility with smart card international standards, like ISO 7816.

A. *Smart Card system and communication model*

A smart card system is composed of hardware and software components. These components are: Support software, software for communication with the card acceptance device (CAD), the CAD itself and the smart cards and their applications.

User-CAD communication software (host application) This software is responsible for the communication between an external application, called "host application", and the code running inside the card. It sends commands for the smart card application and receives the responses to these commands. This software can be included in a desktop computer, in a cell phone or in a security subsystem.

Card Acceptance Device (CAD): A CAD is the device located between the host application and the smart card. It supplies power to the card and is the means of communication between the host application and the application inside the card. A CAD can be connected to a desktop or a terminal, such as an electronic payment terminal.

Smart Cards and their applications: Applications are stored in the card memory. This can be done when the card is being manufactured, installing applications in its ROM memory, or later, installing the applications in the card's non-volatile and writable EEPROM .memory. The EEPROM memory can also be written by applications to store their data. Smart cards also have a (faster) RAM memory to store temporary data. Languages like C, the assembly language of the card and Java Card can be used to develop these applications. Today, Java Card is supported in more than 95% [8] of the cards and is considered the best choice when productivity and security are the main requirements.

Support software: This kind of software provides services to a smart card application. For instance, we could have an application that allows the applet to access a credit card operator service in a secure way.

B. *The Java Card Remote Method Invocation framework*

The Java Card RMI assumes that the host application needs to use a service provided by an applet on the card as an application programming interface (API). This service is specified as a Java interface that extends directly the predefined Java Remote interface (see example in Figure 1). The methods of the corresponding implementation class (see Figure 2) are invokable from a different virtual machine (in this case, the host-side virtual machine). This implementation class needs to be developed in the Java Card dialect. This class shall inherit from the CardRemoteObject class provided by the Java Card RMI framework that provides methods to enable and disable the remote access to objects. The RMI also provides the class RMI Service that translates method invocations to APDU-level communications. Java Card imposes restrictions due to the very nature of the platform it runs on: there is a restricted set of basic data types, no threads, no guarantee that there is a garbage collector either. Also, in order to avoid loss of data consistency on the card, the Java Card framework provides transaction facilities, as well as specific mechanisms to distinguish persistent data (need to be maintained when the power is turned off) from transient data (may be erased when the card is reset).

In the Java Card environment, applications are called applets, and are classes inheriting from the java card.

framework.Applet class. An applet must provide an implementation for the methods install and process. The install method creates the applet by invoking its constructor method and registers it in the Java Card Runtime Environment (JCRE), by invoking the register method. The process method receives the APDU messages of the host application, does the initial processing of these messages, and invokes a method, passing to it the APDU object as a parameter. In Figure 3, we present on an example how the implementation of Remote object can be integrated into an applet and associated with a RMI Service object responsible for the communication with the host-side.

Finally, a reference to the remote object needs to be created on the host-side. The Open Card framework provides functions to get such reference. Once bound to a local object, the RMI is transparent to the programmer.

C. *Elements of Java Card programming*

Programming for a smart card requires special care against two possible problems:

Available memory Smart card usually has a very limited amount of memory; in addition, the runtime environment does not necessarily have a garbage collector.

The programmer needs to apply specific memory allocation strategies. For instance, a method should avoid, at all cost, allocation of objects, as there is no available garbage collection mechanism. Thus, object allocation is usually restricted to the constructor. Also Java Card provides a special exception mechanism, where the cause of an exception is a short value instead of a string as in Java. This mechanism is optimized to avoid multiplicating the number of exception objects in the card memory.

Data coherency The smart card may be physically removed from the card acceptance device at any time, causing a power failure and interrupting the execution of the virtual machine. To avoid that objects get into inconsistent states, Java Card provides a transaction mechanism that guarantees atomicity of execution (at a cost). In addition, objects may be specified as being persistent (maintain their value when power is turned off) or transient (are reinitialized when power is turned off).

V. CONCLUSIONS

The main contribution of this work is to provide support for a rigorous development of Java Card components for smart card aware applications, based on the B method.

This paper presents the effective integration of transaction support in the Java Card. It reports the basic transaction semantics required by the Java Card 2.1 specification which only requires the minimum functionality needed for simple transactional computations. For instance, the Java Card specification and especially its transaction model suffers from its static allocation model where any space allocated within transactions may not be released in case of an abort. In contrast, we have shown that object instantiations can easily be integrated in the transaction mechanism even in case of the tight memory resources on a smart card. The various possible implementation choices are discussed in detail, including various log schemes, their impact on performance and memory usage and possible optimizations.

REFERENCES

[1] D. Deharbe, B. G. Gomes, A. M. Moreira, "Automation of Java Card component development using the B method", *Engineering of Complex Computer Systems, ICECCS*, 2006.

[2] M. Ostriches, "Transactions in Java Card", *Computer Security Applications Conference*, 1999.

[3] C. March, "Verification of JAVA CARD Applets Behavior with respect to Transactions and Card Tears", *Software Engineering and Formal Methods, SEFM 2006*, 2006.

[4] M. S. Jin, M. S. Jung, "A study on how to reduce time and space by redefining new byte code for Java Card", *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, 2005.

[5] D. Robinson, "The worldwide market for smart cards and semiconductors in smart cards", *Technical report, IMS Research*, 2005.

[6] Z. Chen, Addison Wesley, "Java Card Technology for Smart Cards: Architecture and Programmer's Guide", *ISBN: 0201703297, 2000*.

[7] J. R. Abrial. , "The B-Book — Assigning Programs to Meanings". Cambridge University Press - 0521021758.

[8] "Java card technology at-a-glance: The foundation for secure digital identity solutions". SUN MICROSYSTEMS INC, 2005.

