Transactions in Java Card

Marcus Oestreicher
IBM Zurich Research Laboratory
8053 Rueschlikon, Switzerland
oes@zurich.ibm.com

Abstract

A smart card runtime environment must provide the proper transaction support for the reliable update of data, especially on multiapplication cards like the Java Card. The transaction mechanism must meet the demands by the applications and the system itself within the minimal resources offered by current smart card hardware. This paper presents the current transaction model implied by the Java Card 2.1 specification, highlights its shortcomings and presents a detailed discussion of possible implementation schemes and their optimizations. It especially addresses the problem of object instantiations within a transaction in the Java Card 2.1 specification and presents an effective solution.

1 Introduction

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.

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 multiapplication 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 [1]. 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 userlevel transactions in the traditional sense [2]. 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 subtransactions 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.

Section 2 gives an overview over the possible integrations of transactions into the different types of smart card systems, especially into interpreter based systems. Among

them is the Java Card whose execution and memory management model is introduced in Section 3. Section 4 discusses transactions on the Java Card in depth. Section 4.1 presents its transaction API and details its pros and cons. Section 4.2 presents the minimum functionality which is expressed in the Java Card 2.1 specification. However, the Java Card specifications inhibits some problems described in Section 4.3. Section 4.4 explains the transaction implementation options on the card, especially the possible log strategies. Section 4.5 deals with the problem of object instantiations within transactional boundaries and presents a solution. Section 5 finally draws our conclusions and presents future ideas.

2 Approaches in Smart Card Operating Systems

As soon as a smart card is inserted into a smart card reader, an external application can start a communication and send commands to the smart card. The card acts as a server, fulfills the requested operation and returns a reply. A set of basic commands is described in the ISO 7816-4 specification which defines an interoperability standard at the level of the command exchange [3]. A smart card conforming to this specification presents the stored data as a secured file system to an external application. An external application can select files in directories and read and update their data after a successful authentication. The ISO specification does not prescribe a transactional concept for the update of stored files. While individual commands updating data should be executed atomically, a sequence of updating commands needs not to be atomic, especially as extremely memory limited smart cards may not provide the necessary resources for such additional guarantees. Thus the external application is supposed to keep track of and manage consistent states. An ISO file system based card could be extended to provide the atomicity of all requested updates. However, an external application might need a more fine granular control over which records belong to the transaction or not. In this case, new commands for transaction control must be introduced limiting the interoperability of the card.

Some smart cards offer a convenient transaction model in form of a database application or even a database operating system [5]. External applications can access and update the information stored in relational tables by providing sufficient indexing information and authentication. In contrast to filesystem based access, updates are transactional by definition. The database model remains sufficient as long as the necessary data can be easily modeled within a relational table and the application does not rely on specific authentication and encryption schemes.

Multiapplication and post issuance smart cards allow the deployment of many applications on a smart card and the extension of the card functionality by installing new applications at a later time on the smart card. Each application is independently selectable by an external application and is responsible for servicing its requests. In multiapplication cards where machine code serves as the executable content, the applications are fully responsible for providing transactional semantics as applications are allowed to directly access the contents in memory. However, direct write accesses to memory which must be logged during a transaction can in general only be caught by explicit support in the language and compiler.

Control over memory access is a basic benefit of an interpreter as the basic execution engine. Interpreters can easily ensure that different applications only access the parts of the memory which have been assigned to them so far. Other than that, an interpreter can make sure that no memory cell is overwritten during a transaction where its previous content has not been saved for potential restore at a later time. Any computation within an application on the smart card can be part of a transaction and transactional computations can be integrated easily in the programming language.

Smart Card interpreters achieve memory protection in two different ways. Systems like MULTOS realize a software memory management unit where the instructions may refer to memory by address but each access gets guarded and checked towards granted areas [6]. Smart card environments like the Java Card offer the possibility to rely on the protection mechanisms of a type safe language which prevents arbitrary accesses to memory. The referential integrity of the language is preserved. An application is only allowed to access the elements or fields in an object or class. One might use this information to record the changes during a transaction at a higher level and record the operations applied to the individual objects. This provides additional information about transaction failures during future investigations, but increases heavily the information needed for the transaction recording.

3 Java Card Introduction

3.1 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. For instance, the Java Card does not support all primitive types and does not allow the dynamic download of classes [7]. Instead, a converter is used to package all classes of a Java Card application into one executable file and to reduce its size by prelinking it for the execution on the card as far as possible. The converted package can then be downloaded on the card where a Java

Card application, an applet, can be installed in a separate step.

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. From now on, 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 its *deselect()* method.

3.2 Memory Management

The applet instance and associated persistent objects of an application must survive a session. Therefore they are placed in the non volatile storage on a card, usually EEPROM. 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 EEPROM 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 looses 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 [9] 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.

4 Transactions in Java Card

4.1 Language Integration

The described memory model shares its main properties with the Java Card transaction model. The persistence or transience property is orthogonal to the type of an object. Any update of an object can be transactional independent of its concrete type. Other than that, the transaction scheme provides the following features:

1. Persistent updates are independent of transactional updates. Changes to objects residing in EEPROM persist even when occurring outside of transaction boundaries. While a single EEPROM field access has to be atomic regarding to the Java Card specification, multiple writes to EEPROM inside or outside a transaction may differ in their behavior.

2. Transactional independence: Source code executed inside or outside a transaction can look exactly the same.

3. Execution within transactions do not compromise Java security:
   No changes have been applied to the language or to the instruction set. Thus the converter remains independent of the transaction mechanism. The recording of state changes is invisible and unaccessible to the executing applet.

Figure 1 shows the current API in the Java Card specification for initiating, committing and aborting transactions. The control of transactions by static methods has a number of disadvantages. The begin and end of a transaction is not connected to each other, neither in the program text nor at runtime. As a result, the execution state can not be reset to a consistent state when a transaction is aborted by request of an applet using *abortTransaction()*. Instead, execution continues right after the *abortTransaction()* call. Transactional

Figure 1. Transaction API

```
JCSystem.beginTransaction();
JCSystem.commitTransaction();
JCSystem.abortTransaction();
```

systems like Transactional-C extend the language by constructs allowing the linguistic connection between begin,

commit and abort blocks [10]. In case of abort or commit execution continues at well defined locations. PJama achieves a similar effect by expecting the transaction to be coded within one single instance method [11]. The runtime environment will then execute the given method within a transaction and return in any case, commit or abort, from its invocation.

Such a mechanism adds the overhead of one temporary instance per transaction which might still be acceptable even within the resource constrained Java Card environment. However, the encapsulation within one method interferes with the event triggered execution model of an applet in the Java Card 2.1 specification. It is not possible to extend the lifetime of a transaction across multiple commands during a session as soon as the transaction is encapsulated within a method invocation. As a method invocation can only last as long as the invocation of the *process()* method, the transaction boundaries can not be connected with each other as soon as transaction lasts longer than one single command. The current API therefore favors flexibility and resource friendliness although the missing linguistic connection is partly responsible for some of the problems described in Section 4.6.

4.2 Basic Java Card Transaction Model

As soon as a transaction is started, the system must keep track of the changes to the persistent environment. The system must at least record the state before the transaction and the most current value for any given element during the transaction. The updates must be logged at the granularity of a single access. Large transactional systems group objects in pages, manipulate them in RAM during the transaction and log changes lazily at the granularity of the page into stable storage. However, the necessary RAM resources are by no means available on current smart card hardware.

The transaction system must provide two guarantees. If the system commits a transaction on request by an applet, it must guarantee that the changes to the persistent set are applied in any case. Any necessary commit information must be stored persistently at commit time to allow for the restart of the commit process in case of sudden power loss. If the commit process succeeds without a crash, execution continues after the return from the commit method.

Whenever a transactional computation aborts, the system must be able to restore the state at the beginning of the transaction. The reason for an abort firstly includes system crashes, e.g. sudden power losses, or system initiated aborts of applet computations. The system throws an exception in case of any irregularity during the transaction processing, for instance due to a transaction buffer overflow, and may abort the applet computation for instance in case of an exception not being handled by the applet. The system then recovers the previous applet state where the recovery information had to be stored persistently to be able to restart the recovery process in case of a sudden power loss. The system is then free either to deselect the current applet or to let it continue in its current session.

An applet can always explicitly request the transaction abort by an *abort*() method invocation, for instance after it caught an exception thrown by the system. The applet remains selected to be able to react to the abort and to further communicate with the external application.

4.3 Java Card 2.1 Limitations

The Java Card specification expects the recovery process to take place immediately at the invocation of the *abort()* method. The persistent state is brought back to its initial state while the execution state and temporary instances are not affected and applet execution continues after the return from the *abort()* invocation. As the state has been recovered, the applet can for instance immediately try to restart the aborted transaction. As it will turn out, the point in time defined for the recovery process has severe implications on the flexibility of the transaction mechanism. In general, the action and the point in time of the recovery can be varied and still allow the future execution of the same applet during the same session.

Although the Java Card 2.1 specification limits the maximum lifetime of a transaction to the duration of one APDU communication, there are scenarios where computation must be transactional over multiple APDU's. For instance, a download of a new application should be encapsulated within a transaction. Other than required for system relevant processes, applications in general benefit from the extension of the maximum transaction duration. For example, downloads of new keys of arbitrary length can span multiple APDU's, should be possible with the regular transaction mechanism, and should not require additional transaction logic by the application.

Allocations by the system or applications must also be covered by the transaction mechanism. The Java Card specification indeed specifies the installation of applets as a transactional process. All objects which are created during a failing applet initialization must be freed. Other than that, the current 2.1 Java Card specification does not require the release of allocated memory within transaction boundaries in case of an abort. In contrast to standard Java, the Java Card specification assumes all object instantiations to take place at installation time and not at any later time. However, some applications may not know or do not have any real worst case requirement which they could allocate at installation time. A general database or data storage application on a card might want to allocate dynamically as many records as an external application may need [13]. In these

cases the system should limit the application resources, but not the application itself. Especially with a garbage collection scheme on the card and increasing memory capacities, memory releases and new instantiations are easily affordable for applications. The system must then guarantee that no memory is lost when new objects are created during a transaction aborted at a later time. Indeed, reclaiming this memory is already required by the definition of a transaction, but hard to enforce in the resource limited Java Card environment.

Our Java Card implementation tries to avoid native code as often as possible. System services like the secure download of new applications and the update of keys, part of the implementation of the Visa Open Platform specification, are almost completely written in Java [12]. Thus the transaction mechanism can be tailored completely towards the requirements of Java applications and need not be designed to support explicitly processes written in native code. As we rely on the general transaction mechanism for performance sensitive applications like the application download, the transaction implementation must be runtime efficient. Other than that, the transaction implementation must take the scarce resources on the card into account and be space efficient.

4.4 Old versus New Value Logging

Updates or writes to the persistent set occur within the interpreter loop only on the access of persistent instance fields, static fields and arrays. A second source are native methods which must use special access operations to not bypass the transaction mechanism. Especially the native *Util.arrayCopy()* methods allow the transactional update of a number of array elements at once [8].

Two schemes are well known for the logging of write accesses during a transaction, e.g. either new value or old value logging [2]. In case of old value logging, the update of a location during the transaction occurs in place, e.g. directly at the referenced location. The general properties of old value logging are:

- fast read accesses as the up-to-date values are always stored at the referenced location.
- the original value for a given location must be saved in a transaction buffer, typically once at the time of the first write access to the location.
- committing a transaction is cheap as the new value are already in place.
- aborting a transaction is expensive as the saved values have to be written back to the original locations.

In case of new value logging, each value for a store operation to a given location is saved in the transaction buffer during the transaction while the original value remains at the affected location. The general properties are here:

- a slow read access as the up-to-date value for a location must be searched in the buffer.
- write operations always have to update the buffer as any new store operation has to be recorded there.
- committing a transaction is expensive as the new values have to be written to their target locations.
- aborting a transaction is cheap as the original values are still in place.

Although the advantages and disadvantages still apply in general in case of the Java Card, their degree depend on the exploitation of the memory characteristics found on a smart card. For instance, the performance aspect depends here mostly on the number of necessary single or block EEPROM writes whereas accesses to RAM are negligible to a large extent. One might also include the typical access pattern of Java Card applications into account where writes to the same location during a transaction are usually rare. So what are reasonable implementations and the achievable performance for both schemes on a Java Card ?

Old Value Logging

Read performance always remains excellent in case of old value logging. In case of a write, the referenced location has to be checked for having already been saved. A reasonable implementation for current smart cards scans the transaction buffer linearly for the given location and if found, the write succeeds directly to the target location. As multiple updates of the same location are rare, the best case for the write performance -one single EEPROM write - does not occur too often. If the former value of the given location has not been saved so far, a new entry consisting of location and original value must be added **persistently** to the transaction buffer to support a recovery process in case of sudden power loss.

Two schemes are conceivable, a mark or counter based transaction buffer scheme. The latter one adds first the new entry to the buffer and then increments the entry counter of the buffer. The counter must be incremented atomically, for instance with the help of a shadow counter and a flag indicating which counter is currently valid. Thus three EEPROM writes are necessary. One block write for the new entry, one write for the incremented shadow counter and one write for flipping the counter flag. Performance can be increased with the mark scheme where a flag after the last entry in the buffer indicates its end. Entries are added to the

buffer by first appending the new entry with the new end marker in a single block writte and then clearing the previous end marker in a second single EEPROM write access. The number of EEPROM accesses is reduced to two while the entry size is increased by an additional byte.

Table 1 summarizes the properties of a old value logging scheme with a marked buffer implementation. Appending a new entry needs two EEPROM writes. In case of commit, the expected total number of EEPROM writes per location is then expected to consist typically of three assuming multiple updates of the same location are rare; two for adding an entry, one for updating the target location.

In case of commit, the transaction buffer must just be marked invalid and the transaction is completed. In case of abort, the saved values in the buffer are written back to their former locations. After a sudden power loss the write process may just be restarted from the beginning of the buffer as locations and values in the buffer remain constant and thus can be rewritten as often as possible (although the number is actually limited by the physically possible number of EEPROM writes).

Table 1. Logging Scheme Comparison

Logging Strategy	New Value	Old Value
Commit Costs	High	Minimal
Abort Costs	Minimal	High
Minimum E2 Accesses for Logging	1	2/Log Entry
Maximum E2 Accesses for Logging	1/Store	2/Log Entry
Expected E2 Accesses per Committed Store	1 + 1/Store	3/Store
Expected E2 Accesses per Aborted Store	1	3/Store
Writes per Log Entry on Abort	0	1

New Value Logging

Similar overall performance can be achieved in the new value logging scheme dependent on the implementation and the available resources. Read performance lags always behind as the transaction buffer must be scanned - typically linearly - for a formerly written value. The situation can be better in case of the much more expensive write operations. A straightforward solution will scan the transaction buffer for a formerly written entry for the given location and replace its value with the new value in a single EEPROM write operation. If the location is accessed the first time - the most common case for typical Java Card applications - a new entry must be added to the buffer **non atomically**, e.g. with one single EEPROM block write. Thus the performance can be increased significantly if the transaction buffer is cached in RAM and written out lazily to EEPROM on overflow. If the RAM resources are not too limited and the transaction does not involve too many write operations, all memory accesses can be logged within the cache in RAM and are only written to EEPROM in one single EEPROM block write at commit time. The buffer must then be saved persistently as any started commit operation has to be completed after a sudden power loss at the time of the next card reset. The runtime environment will then scan through the transaction buffer and apply the stored values to the given locations. Aborts are again free in the sense that the contents of the transaction buffer can just be discarded.

Table 1 summarizes the properties of a new value logging scheme with RAM caching. Best commit performance can be achieved if all log entries can be cached in RAM and all entries are saved at commit time in EEPROM with *one* single block write. The value in *each* entry must then be flushed to its target location with another EEPROM write. In the worst case however, EEPROM has to be accessed on each log operation for instance if log entries are reused and an entry for a given location is already existent in EEPROM.

Other than pure performance, the necessary memory resources are another key aspect for choosing the right logging scheme which are for instance high in case of a cache based new value logging scheme. However, there is still another general advantage of the new value logging scheme which arises from the fact that an abort or a sudden power loss is more likely to occur during the application processing than during the commit or abort process by the system. Thus someone might choose the new value logging scheme in general as it reduces the amount of work for the recovery process in case of an application abort drastically.

4.5 Object Instantiations within Transactions

The Java Card specification does not enforce possible object instantiations outside of the installation method. Other than that, it also explicitly states that object allocations within transactions may fail and any allocated space is allowed to get lost forever in case of an abort [8]. Clearly, this does not conform to proper transactional semantics where the state of the applications and the system is expected to be exactly the same as before the transaction in case of an abort and thus any allocated space in between is released. This is especially very harmful as there is a practical need for object instantiations outside of the applet installation method and under a proper transaction control.

For instance, our Visa Open Platform implementation relies completely on a real transaction mechanism for the download of new applications [12]. During the transaction, a new array is created and the executable content is

downloaded and stored in the newly allocated object. If the transaction fails, the transaction mechanism ensures that the newly allocated object will go away during the abort process. Indeed, if any change to the persistent memory is included in the transactional mechanism, including the changes by the system to the heap management structures etc., the persistent state is recovered completely in case of an abort and any newly allocated object is automatically released.

However, there are a few remaining problems. The newly allocated array for the code to be downloaded in the given example can be huge and as any write to the array incurs an additional entry in the transaction buffer, the buffer is likely to overflow during the transaction. However, each access to a newly created object can be easily detected within the interpreter and directly forwarded to the contents of the newly created array. In the object aware Java bytecode, objects are always addressed by an object reference and offset. When the object header and its heap management information is logged at instantiation time, the transaction mechanism can decide on each access whether a given object already existed before or has been allocated during the transaction. The referenced object is just searched in the buffer and if it is not found, the object already existed before and the store operation is regularly logged. If it is found, it has been newly allocated and the value can commence directly within the newly allocated region at the given offset. In case of commit, the object header and its heap management information is written permanently, the object thus becomes allocated persistently. In case of an abort, it is automatically released. This optimization can therefore reduce extremely the necessary transaction buffer size during a transaction.

The most hindering problem is the fact that the transaction mechanism logs only writes to persistent fields. Thus, temporary references stored in RAM may still reference the newly instantiated objects after an abort. As long as these references exist, the virtual machine can not release the referenced objects. If the runtime releases the objects, it must reset the relevant references to a defined state. The most simple approach for avoiding this problem is to deselect an applet immediately in case of an abort and to recover the persistent set. However, if an applet gets automatically deselected, it depends completely on the external application to reselect and reactivate it again.

There are two potential sources for references within RAM to the areas of aborted object instantiations. First, an application might have stored such references in transient objects. These can be found and reset by a RAM garbage collector [9]. It has just to be adapted to search for specific persistent objects and reset the referencing location in RAM. Other than that, local variables in the current execution frame may contain such references in case of an abort due to the missing linguistic connection between the transaction boundaries. Figure 2 shows a code example where f is a local variable which is still accessible after the application initiated an abort and still refers to the newly instantiated object. If the semantics for *abortTransaction()* are

Figure 2. Problematic JCSystem.abort()

```
Foo f;
JCSystem.beginTransaction();
...
f = new Object();
...
JCSystem.abortTransaction();
f.doIt();
```

defined to recover the state and release the allocated objects immediately, the references on the stack must be reset, too. Which elements on the stack are references can be gathered practically in two ways. Firstly, the interpreter may implement a type tagged stack where each stack slot is marked with its type. This allows the reset of problematic references immediately in case of abort, but reduces the interpreter performance in general and increases the size of the runtime stack. Secondly, instructions can be checked lazily at execution time not to operate on invalid references and throw an exception in case. However, this still introduces a performance penalty and especially makes it very hard to reuse and reallocate the space for an aborted object instantiation as the system must ensure that no other reference to this area still exists. This seems to be the main reason why the Java Card specification allows memory to get lost in case of allocations within transactions.

The restrictions on the interpreter implementation can be reduced when the point in time for the recovery process is delayed until the applet returns from its *process()* invocation by the system. The stack is then unwound and only the temporary objects have to be scanned for problematic references. The applet is then limited in so far that it can not immediately try to restart the transaction, but must wait for another command by the external communication. However, we expect an applet to return an error code in case of a failure anyway and wait for new commands for further processing. Instead of restricting the system with an expensive and fixed interpreter architecture, we propose instead a small limitation on the possible communication behavior of Java Card applications. We therefore suggest that *abortTransaction()* throws an exception by default to remember a programmer that his applet is going to operate on still unrecovered data and will be recovered on return from the current applet invocation.

5 Conclusion And Future Work

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.

An extended transaction mechanism can be used by a wide range of applications, for instance by system services like the download of applications, or by applications to download and update arbitrary data like keys. It is also used by applets to reliable audit the progress of computations. Java Cards do not provide enough memory resources and symbolic information to allow a general audit of application processes by the system. Thus, current applications need to record any audit information by themselves. In the future, we want to extend our transaction mechanism to provide a standard audit mechanism which can be used by a broad range of applications. However, as specifications like Visa Cash or Geldkarte can not rely so far on a standard audit service, they specify their own and different audit mechanisms which then have to be implemented by the applications themselves [4].

The described application scenarios are already fully supported with the transaction mechanism proposed in this paper and the Java Card platform therefore provides a flexible and reliable platform for smart card applications.

References

[1] Jim Gray, The Transaction Concept: Virtues and Limitations, Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings

[2] Jim Gray, Andreas Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann 1993, ISBN 1-55860-190-2

[3] ISO/IEC 7816-4, Identification Cards - Integrated circuit(s) cards with contacts - Part 4: Interindustry commands for interchange, 1995, ISO/IEC 7816-4:1995(E)

[4] Leo van Hove, A selected bibliography on electronic purses, http://cfec.vub.ac.be/cfec/purses.htm

[5] E. Dufresnes, P. Paradinas, J.-J. Vandewalle. CQL, a Data Base in Smart Card for healthcare Applications, Height World Congress on Medical Informatics, Edmonton Canada, July 1995

[6] Scott Guthery, alt.technology.smartcards FAQ, 1998, http://www.scdk.com/atsfaq.htm

[7] Sun Microsystems Inc., Java Card 2.1 Virtual Machine Specification, Final Revision 1.0, March 1998, http://java.sun.com/products/javacard/JCVMSpec.pdf

[8] Sun Microsystems Inc., Java Card 2.1 API Specification, Final Revision 1.0, March 1998, http://java.sun.com/products/javacard/htmldoc/index.html

[9] Marcus Oestreicher, Krishna Ksheeradbhi, Object Lifetimes in Java Card, USENIX Workshop on Smart Card Technology, May 1999

[10] Paul Taylor, Transactions for Amadeus, Thesis, Department of Computer Science, Trinity College Dublin, August 1993

[11] Atkinson M.P., Jordan M.J., Daynes L., Spence S., Design Issues For Persistent Java: a type-safe, object-oriented, orthogonally persistent system, Seventh International Workshop on Persistent Object Systems, February 1996

[12] Visa International, Open Platform Main Page, http://www.visa.com/nt/suppliers/open/main.html

[13] RSA Laboratories, “PKCS #11: Cryptographic Token Interface Standard”, December 1997, RSA Data Security, Inc., http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-11.html