Checkpoint Model

1. Introduction

The current Java Card computing environment includes a mechanism for object persistence. Such a mechanism is still desirable in the next generation Java Card system. This document describes a proposal for such a mechanism, namely, the *checkpoint model*.

Section 2 presents an overview of the checkpoint model and OPJ[1], another persistence system for Java from which the checkpoint model is derived. Section 3 presents a brief summary of the relationship between persistence and sharing. Section 4 presents the application programming interface of the checkpoint model and finally section 5 discusses some of the other proposals that have been put forward for the persistence mechanism in the next generation Java Card system.

2. Overview

The checkpoint model is closely related to the Orthogonal Persistence for Java (OPJ) specification, which was drafted by a group at Sun labs. The OPJ specification is guided by three simple principles:

- **Type Orthogonality**: persistence is available for all objects irrespective of type
- **Persistence by Reachability**: the lifetime of each object is determined by reachability from a set of *root* objects.
- **Persistence Independence**: code is identical whether it is operating on short-lived or long-lived objects.

While the OPJ specification mostly succeeds in holding to these principles, it makes certain exceptions to deal with issues that would be impossible to address otherwise. For example, the principle of persistence independence cannot be applied to classes that represent some state external to the virtual machine (such as classes for file system input and output).

[1] http://www.sunlabs.com/research/forest/COM.Sun.Labs.Forest.doc.opjspec.abs.html

Similarly, the checkpoint model limits its adherence to the three principles as well to take into account certain characteristics of the platforms on which Java Card is intended to run. These characteristics are:

- A volatile[1] random access memory (RAM) memory system.
- A non-volatile memory (NVM) memory system.
- Writing to NVM is an order of magnitude (at least) slower than writing to RAM.
- Reading from NVM is comparable in speed to reading from RAM.
- RAM is very limited while NVM is relatively large (usually at least 8 x RAM).
- Power failures can occur at any time.
- The integrity of the objects in the NVM must be preserved across power failures.

Based on these characteristics, a reasonable Java Card implementation could be expected to display the following properties:

- The NVM is used as an object store where the objects can be accessed directly at runtime without having to be re-formatted. This is in contrast to using the NVM as a file-based persistent storage system.
- Making RAM copies of objects in NVM is avoided where possible.

The components of the OPJ specification that are adopted (either partially or completely) by the checkpoint model include persistence by reachability, persistence of static variables, the persistence transactions. These parts of the OPJ specification are reproduced below (in blue font) followed by a description of how they are adopted by the checkpoint model. Also included is a brief discussion of why thread state is not persistent in the checkpoint model.

2.1 Persistence by Reachability

An object is *persistence reachable* if it is reachable by the standard rules for object reachability in the Java language, *unless* it is reachable solely through chains of references that pass through variables marked by the `transient` modifier. An object is made persistent at a checkpoint if it is persistence reachable. This is intentionally not “if and only if”. An implementation is encouraged to avoid making unreachable objects

[1] Volatile means that the data in the memory is lost when the power is removed.

persistent, but it is not required to do so, as such objects will eventually be garbage collected.

A class `Class` instance is persistence reachable if it is directly referenced under the standard rules for object reachability, as described above, or is referenced by a `ClassLoader` instance that is itself persistently reachable, if at least one of its instances is persistently reachable, or if some array of its instances is persistently reachable. The Java Language Specification[1] states that the *bootstrap classloader* instance is always reachable. It follows that all classes loaded by the bootstrap classloader (bootstrap classes) are persistence reachable, as are all the objects that are persistence reachable from static variables in those classes.

Any *resolved*, symbolic reference to a class C in the definition of some class CC will ensure that if CC is persistence reachable, then so is C. A symbolic reference exists from CC to C if C is the super-class of CC, or if C implements C, if C is used as the type of a variable (static, instance or local) or as the type of the parameters or result of a method or constructor, or if CC uses a static variable of C. Note that this constraint does not restrict an implementation in its choice of eager or lazy resolution of symbolic references. However, once resolution has occurred the binding must be maintained.

The `transient` modifier can act to prune the reachability graph. On a resumption, and before any active use of an object, all member variables marked `transient` are reset to their default values. An implementation is expected to exploit this to exclude objects that are reachable only through transient variables in the set that is made persistent, but this is not a requirement.

The checkpoint model partially adopts this definition of reachability. The main difference is in the definition of persistent roots.

Firstly, as there are no class loaders in Java Card, the reachability of `Class` instances is simplified to mean all classes in the system. That is, every `Class` object is a root. In particular, every static variable is a root (discussed further below).

Secondly, thread state is not persistent (see following section) and as such, local variables are not roots of persistence.

[1] http://java.sun.com/docs/books/jls/

2.2 Static variables

The use static of variables with respect to persistence varies. For some applications, a static variable can contain a value that is crucial to the invariant of some class. For example, a class that hands out sequential, unique, integers, using a static variable to hold the current value, will fail to uphold its contract unless the static variable is made persistent with the class. To ensure the correctness of such applications, static variables are made persistent in the checkpoint model. Other applications may use static variables that are inherently transient. These applications may make use the mechanism of declaring such variables to be `static transient`.

Making static variables persistent by default also has two other benefits. Firstly, they provide a convenient definition for the roots of persistence. That is, all non-transient static variables can be considered as persistent roots[1]. Secondly, persistent static variables prevent the troublesome issue of whether or not to re-execute the static initializers for classes that have persistent instances. The Java language semantics dictate that the initializer for a class must be executed before any instance of that class can be created.

2.3 Transactions

Both OPJ and the current Java Card specification (2.2) conform to the standard ACID transactional properties – namely *atomicity, consistency, isolation* and *durability* – with respect to the persistent memory. The primary difference is in the granularity of the computation that is considered to be a single transaction. OPJ defines a transaction as the unit of virtual machine computation between two suspended computations in a persistent store. Java Card 2.2 on the other hand, provides for much finer grain transactions with explicit APIs for starting, committing and aborting transactions. It also includes the notion of auto-transactions. These are updates to persistent objects that occur outside the scope of an explicit transaction.

The checkpoint model for transactions more closely models the coarse grain model of OPJ than the existing Java Card 2.2 model. A transaction in the checkpoint model is defined as the point between two checkpoint calls.

While this may appear to be a step backward from the existing Java Card transactional model, it actually avoids all the problems of ensuring the ACID properties for fine grain transactions in a system that will also support true multi-threading. The current Java Card

[1] In a typical Java Card implementation, most system defined static variables are immutable and hence only pay the price for being persistent once. The belief is that the same will hold true for most application defined static variables.

transactional model only manages to support the ACID properties due to a single-threaded, event driven application model. That is, isolation and consistency are currently guaranteed by the fact that there simply is no contention for access to the persistent store between separate threads. Given that the Java Card system is moving towards a multi-threaded model, it is easier to provide partial support for transactions and leave the rest up to the programmer who will therefore have more flexibility in deciding the type of transaction model to apply. For instance, it may be desirable to allow sharing of persistent data between two threads while they are both in a transaction if they are known to cooperate correctly.

2.4 Managing external state

OPJ provides a mechanism for managing program state that is not represented using the Java programming language. For example, a Java class implementing file input may simply hold onto a file descriptor of the open file. All operations on the file are implemented as native calls to operating system file operations that take this file descriptor (and other relevant parameters). That is, the state of the open file is external to the VM and the file descriptor is simply a proxy for this state.

It is questionable whether classes that represent some external state should be persistable in Java Card. For example, all objects used for IO would presumably be transient. However, if it was deemed necessary to support persistence for classes with external state, then an adaptation of the OPJ runtime event mechanism could be employed.

The call back mechanism for managing external state can also be used to provide a way for objects and classes to re-initialize their `transient` variables upon VM resumption.

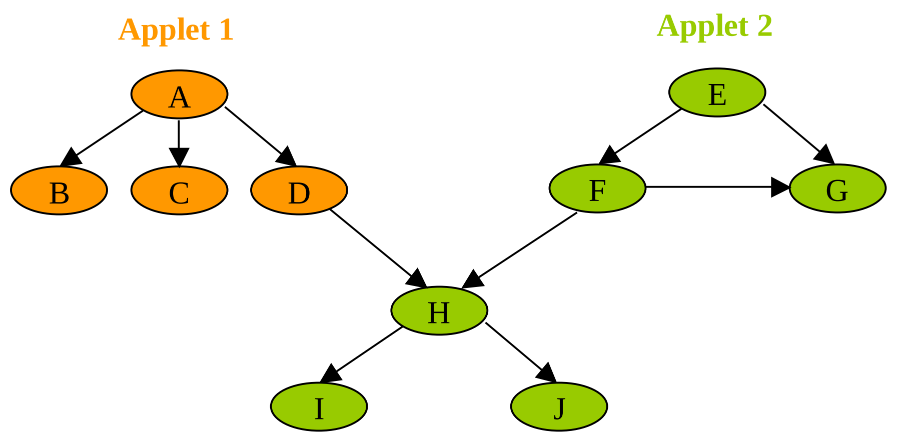
2.5 Persistence of thread state

While OPJ specifies that thread state and all instances of `java.lang.Thread` can be persistent, doing so in a Java Card environment is difficult. For efficiency reasons (as well as possible security concerns), live thread state is typically stored in RAM. To persist thread state would require significant amounts of data to be copied between NVM and RAM at checkpoint/resume points. In addition, in the Java Card world, it is not clear that persistent threads would simplify the established application model. As such, the first principle (type orthogonality) is limited in the case of the `java.lang.Thread` class by excluding it from the types that can be made persistent.

3. Sharing and persistence.

3.1 Checkpointing shared graphs

The checkpoint model must account for persistent object graphs that include objects shared by another persistent object graph. For example, consider the following graphs of objects:



The graph rooted by object H and owned by Applet 2 is shared by the graphs for applets 1 and 2. Under this scenario, if the graph rooted by object A is checkpointed, then the subgraph rooted by object H will also be checkpointed. The same is true if the graph rooted by object E is checkpointed. However, the graph rooted by object H can be checkpointed independently by applet object H. That is, the properties of persistence and shared graphs are:

- the state of a shared object graph can be independently checkpointed by the applet owning the shared object graph itself
- the state of a shared object graph can also be checkpointed by the client of the shared object graph by virtue of reachability

3.2 Firewall considerations

The existing firewall mechanism is sufficient for enforcing inter-applet security. The use of interfaces and proper object encapsulation provides all the functionality required to share objects across firewall contexts. The issue of not being able to share arrays is only a matter of inconvenience for insecure code. If array data must be accessed across a firewall context, a wrapper class can be used. This enables both finer control of access to the elements of the array (e.g. they could be read only) and does not require additional mechanisms to be added to the system.

With respect to checkpointing, there may be some need to consider adding a check when a checkpointed graph crosses a firewall context. While it is impossible for such an inter-applet link to be established without the co-operation of both applets, such a link may be established as the result of a hardware attack.

4. API

Since persistence is a core part of any Java Card system, the most appealing way to specify the checkpoint model API is to augment the standard classes, for example, `javacard.framework.JCSystem`, and for simplicity and clarity this specification takes that approach. However, the checkpoint functionality could just as easily be presented as a third-party API.

4.1 The `javacard.framework.ResumeListener` class

This is the interface implemented by a class that wishes to be notified when an application is resumed.

```
public interface ResumeListener {
   public void resume();
}
```

This method is invoked when an application is resumed. In the Java Card context, this corresponds to an applet being selected. This method is also invoked for an object when its state is being rolled back as a result of a call to `JCSystem.rollback` (see below).

4.2 The `javacard.framework.JCSystem` class

The following methods are added to the class:

```
     public static void checkpoint(Object root);
```

This method takes the set of persistent roots which is composed of `root` and all static variables and ensures that these roots and the graph(s) of objects reachable from them are in NVM, copying them from RAM if necessary. The reachability graph(s) is broken by any variables marked `transient`.

```
     public static void rollback(Object root);
```

This method traverses the reachability graph rooted at `root` and for each NVM resident object it finds in the graph it reverts its fields back to the state they were in immediately after they were last checkpointed. Note that for any object, `obj`, in the graph (apart from `root`) this state may not be the same as it was when `root` was last checkpointed given that `obj` may be part of another object graph that was checkpointed subsequent to the time at which `root` was checkpointed.

```
public static void addResumeListener(ResumeListener l);
```

This method adds `l` to the set of registered resume event listeners. It is a no-op to add the same listener more than once. This method typically should be called from the constructor for the listener object.

```
public static void removeResumeListener(ResumeListener l);
```

This method removes `l` from the set of registered runtime event listeners. It has no effect if `l` is not registered.

5. Critique of other proposals

This section discusses some of the other proposals for a persistence mechanism for the next generation of Java Card.

5.1 Trusted Logic proposal

This proposal aims to encompass many of the aspects of existing standards. It draws upon the Java Data Objects (JDO) specification, the OSGi specification as well as aspects of the J2SE standard. While this is a laudable goal in that it attempts to leverage existing standards, the result is overly complex and very hard to understand. For example, it violates a clean separation of concerns by including in a single class (`ObjectManager`) methods pertaining to persistence as well as methods pertaining to access control.

JDO is based on assumptions about the operating environment that are fundamentally different from those of a Java Card. The former is designed for an environment where the persistent storage is separate from the object memory system of the virtual machine. It requires a mapping between object graphs in the virtual machine and the storage format used by the storage mechanism (e.g. database entries, XML, etc). While JDO attempts to provide an object based abstraction layer above some type of secondary storage (e.g.

databases or file systems), it is permeated throughout with concepts that expose its intricate connection with such a non-unified secondary storage system. For example, the API contains classes and methods for making connections with storage systems, tweaking the interaction with these systems, and it even includes a separate embedded query language that is heavily influenced by traditional database access mechanisms.
In contrast, a Java Card is not intended to provide an object based interface to a database backend. Instead it encapsulates a completely closed system of data and code that can interface to external systems.
Thus, models such as JDO (as well as the Serialization based approach investigated previously by Sun) suffer from the following issues and therefore unsuitable for a small platform:

- RAM copies of persistent objects are recreated for each new session. Creation of an object requires the execution of its constructor and possibly its associated class initializer.
- The object heap space is separate from the storage database and requires duplicated resources.

In addition, using the security model of the Java 2 platform seems like overkill for Java Card. In general, the simpler a security mechanism is, the easier it is to deploy correctly and hence the stronger the security guarantee.

5.2 Object Promotion – GemPlus proposal

The Object Promotion model promotes an object from RAM to EEPROM when the object becomes reachable from an object already in EEPROM. It does not provide transaction semantics as such and relies on the Java Card 2.2 transaction model. In addition, it needs to be augmented with a weak reference like mechanism to allow objects in EEPROM to reference temporary objects in RAM (without automatically initiating a promotion operation).
The following is a list of undesirable characteristics:

- The applet programmer may inadvertently promote a temporary object into EEPROM and thus experience performance degradation on updates.
- It relies on the simple Java Card 2.2 transaction model, which may be unsuitable for a more sophisticated application model where there may be contention for an object from multiple independent threads.
- Session key objects may be stored in EEPROM without the explicit knowledge of the applet programmer and may expose a security risk.

- The weak reference adapter (TransientRef) mechanism to reference temporary objects is not very flexible and awkward to use. It does not support temporary primitives[1].

[1] This critique is based on the object promotion model proposal that uses the `TranientRef` class instead of the `transient` keyword.