

Modèles de mémoire en Java Card™*

Introduction du concept de pré-persistance

Serge CHAUMETTE, Damien SAUVERON†

{serge.chaumette,damien.sauveron}@labri.fr

LaBRI, Laboratoire Bordelais de Recherche en Informatique
UMR 5800 – Université Bordeaux 1

351 cours de la Libération, 33405 Talence CEDEX, FRANCE.

Résumé : Les mémoires d'une carte à puce sont des ressources très limitées et qui suivant leur type ont des propriétés bien spécifiques. Aussi pour adapter le langage Java à ces contraintes, les spécifications Java Card ont décrit un modèle de mémoire basé sur les concepts de *persistance* et de *transience*. Le problème est que ces concepts tels qu'ils sont spécifiés, sont sujets à de multiples interprétations pouvant conduire à des modèles de mémoire totalement différents et donc à des implantations aux propriétés elles aussi différentes. Déjà en 1999, des travaux [Oestreicher, 1999b] proposaient d'introduire la notion d'*environnement transient explicite* afin d'obtenir un modèle de programmation plus simple et homogène vis-à-vis des concepts de persistance et de transience. Malheureusement leurs propositions n'ont pas été retenues et aujourd'hui encore, le langage Java Card a une sémantique très différente et bien plus complexe que celle du langage Java. Cet article introduit la notion de *pré-persistance* en vue d'étudier les différentes interprétations possibles des spécifications. Ces travaux ont été conduits au sein du projet *Sécurité Java Card*¹ menés dans le cadre de la collaboration entre le LaBRI et la société SERMA Technologies.

Mots-clés : Carte à puce, Java Card, mémoire, sécurité, systèmes et informatique embarqués.

1 INTRODUCTION

La technologie Java Card [Chen, 2000, Sun, 2003, Sauveron, 2001] a été conçue afin d'apporter aux programmeurs d'applications embarquées sur cartes à puce tous les avantages offerts par le langage Java [Gosling, 2000, Arnold, 2000]. Ainsi adapter l'environnement Java aux cartes à puce, consistait à répondre à la question : « Comment intégrer Java sur un périphérique de nature persistante ? ». En effet, grâce à la technologie des mémoires persistantes qu'elle embarque, la carte à puce peut stocker de façon sécurisée des informations mais aussi les conserver une fois hors tension. Cependant l'environnement Java étant un environ-

nement de programmation temporaire, pour satisfaire aux contraintes des cartes à puce les spécifications Java Card ont choisi d'adopter les concepts de *persistance* et de *transience* issus des systèmes persistants orthogonaux [Atkinson, 1995].

L'objectif de cet article est de présenter les ambiguïtés des spécifications Java Card et donc les choix technologiques auxquels est confronté un développeur. Pour cela nous introduirons la notion de *pré-persistance* afin d'expliquer les différents modèles de mémoire que nous envisageons. Les problèmes que nous présentons sont ceux auxquels nous avons eu à faire face lors de la réalisation de notre plate-forme *Jcatoools* [Chaumette, 2003] qui est une implantation extensible et modulaire des spécifications Java Card.

On notera que nous avons choisi de créer le néologisme *transience* et ses dérivés pour exprimer les propriétés de temporalité spécifiques à certaines données de Java Card. En effet la traduction simple du terme anglais *transient* par les termes de « temporaire » ou « transitoire » n'exprimait pas exactement les concepts sous-jacents.

L'article présentera en section 2 les mémoires physiques des cartes à puce, en section 3 les cycles de vie de la JCVM et des applications Java Card et en section 4 les objets existants en Java Card. La section 5 synthétisera les concepts de *persistance* et de *transience* tels qu'ils apparaissent dans les trois volumes des spécifications. Puis la section 6 exposera les mécanismes d'atomicité et de transactions de Java Card ainsi que l'incidence de la nature des objets sur ces mécanismes. La section 7 décrira les aspects relatifs à la gestion de la mémoire. Ensuite dans la section 8 nous mettrons en exergue deux problèmes des spécifications, nous définirons l'espace *pré-persistant* et nous présenterons les différents modèles de mémoire possibles, leurs avantages et leurs inconvénients. Enfin en section 9 nous confronterons nos travaux avec ceux déjà existant puis en section 10 nous conclurons.

2 LES MÉMOIRES PHYSIQUES

Une carte à puce dispose de trois types de mémoire.

La ROM (Read Only Memory) est une mémoire persistante, non modifiable, qui sert à stocker le COS (Card Operating System) et les données permanentes. Elle est programmée en usine (lors du processus de masquage) et a une taille variant de 32 à 64 Ko.

*Java and all Java-based marks are trademarks or registered trademarks of Sun microsystems, Inc. in the United States and other countries. The authors are independent of Sun microsystems, Inc.

†LaBRI et CESTI de SERMA Technologies.

¹Labélisé *PROGramme Société de l'Information* et financé par le Ministère de l'économie, des finances et de l'industrie

L'EEPROM (Electrical Erasable Programmable Read Only Memory) est également une mémoire persistante qui, contrairement à la ROM, peut être modifiée par une application. Sa taille varie de 24 à 64 Ko. Le problème majeur de cette mémoire est sa durée de vie limitée en nombre de cycles d'écriture (100000 cycles) et en temps de rétention des informations (10 ans).

La RAM (Random Access Memory) est une mémoire volatile et rapide (200 fois plus rapide que l'EEPROM en écriture). Elle est utilisée comme espace de stockage temporaire pour des données fréquemment mises à jour (comme la pile d'appels), ou pour des données temporaires nécessitant un fort degré de confidentialité (comme des clés cryptographiques de chiffrement de session). Sa taille varie de 1 à 2 Ko.

La surface d'une puce étant réduite à environ $25mm^2$, les fondeurs doivent pondérer le nombre d'octets attribués à chaque type de mémoire en fonction de l'espace qu'elle occupe et de l'utilité de ses caractéristiques au bon fonctionnement de l'OS. Par exemple, une cellule d'EEPROM occupe quatre fois plus d'espace qu'une cellule de ROM et une cellule de RAM quatre fois plus d'espace qu'une cellule d'EEPROM.

Par ailleurs on notera que de nouvelles technologies de mémoires persistantes commencent à être utilisées comme la Flash et la FRAM (Ferroelectric Random Access Memory). La FRAM est la plus prometteuse tant en terme de rapidité des écritures (seulement 10 fois plus lente que l'écriture en RAM) qu'en terme d'endurance (100000 fois plus de cycles que l'EEPROM).

3 CYCLES DE VIE

Le cycle de vie des différents systèmes/données présents sur une carte à puce dépend de leur localisation dans les mémoires.

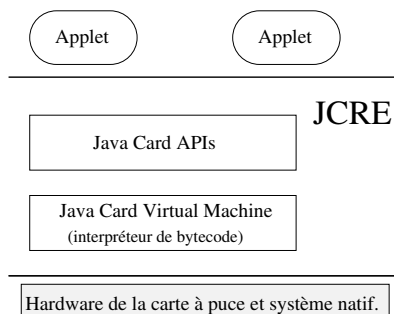


FIG. 1 – Architecture de la Java Card.

De par l'architecture de la plate-forme Java Card (cf. FIG. 1), la ROM contient le système d'exploitation et la machine virtuelle. La RAM quant à elle renferme les différentes structures de données nécessaires à la machine virtuelle Java Card (JCVM) pour fonctionner (p.ex. la pile d'appels). L'EEPROM est utilisée pour stocker les applications et les objets persistants.

Ainsi sur la plate-forme Java Card les informations de la machine virtuelle sont préservées au travers des sessions de communication avec le lecteur et donc contrairement à

la machine virtuelle Java classique utilisée sur une station de travail qui voit son cycle de vie limité à une exécution, la JCVM possède un cycle de vie identique à celui de la carte.

Les applications Java Card, aussi appelées *applets*, sont programmées en langage Java Card, un sous ensemble du langage Java, et utilisent des APIs standardisées. Par exemple toutes les applets doivent hériter de la classe `Applet`. Elles peuvent être chargées tout au long de la vie de la carte sous la forme d'un paquetage au format CAP. À l'instar du format de fichier `class` pour la plate-forme Java, le format de fichier CAP est le format standard de fichier pour la compatibilité binaire pour la plate-forme Java Card. Il renferme donc une représentation binaire exécutable des classes d'un paquetage Java. Une fois le paquetage chargé, la JCVM appelle la méthode `install` pour allouer les ressources requises et enregistrer – via la méthode `register` – un objet persistant, l'instance de applet, auprès de l'environnement d'exécution Java Card (JCRC) pour permettre les futures requêtes à cette applet. Le JCRC implante un mécanisme de pare-feu entre les applets afin de protéger le code et les données de chaque applet d'attaques en provenance d'autres applets. Du fait de la nature serveur des cartes à puce, l'exécution de l'applet se déroule seulement lors de sessions avec un lecteur. En effet, lorsque la Java Card est insérée dans un lecteur elle est mise sous tension, puis le JCRC est initialisé et se met en attente d'un ordre sur le port de communication géré par le système d'exploitation sous-jacent. Ainsi lorsqu'une application externe (*i.e.* le client) initiera une session avec une applet présente sur la carte (*i.e.* le serveur) en envoyant une commande de sélection de l'applet, le JCRC invoquera la méthode `select` de cette applet et la marquera comme active si la méthode retourne sans erreur. Si l'applet est active le JCRC transférera toutes les prochaines commandes envoyées par le client à la méthode `process` de l'applet, sauf les commandes de sélection qu'il interprétera. Ensuite l'applet traitera la commande et préparera la réponse à envoyer au client à la fin de son exécution. Après le traitement de plusieurs commandes, la session de l'applet se terminera soit parce que la sélection d'une autre applet sera demandée et dans ce cas l'applet active sera tout d'abord désélectionnée via sa méthode `deselect`, soit parce que la carte sera retirée du lecteur et dans ce cas il s'agit d'une désélection implicite puisqu'à la prochaine mise sous tension le JCRC désélectionnera l'applet précédemment active. Ainsi une session avec une applet dure depuis sa sélection jusqu'à sa désélection. Les paquetages et les instances des applets peuvent être effacés, et à défaut leur cycle de vie est celui de la JCVM.

4 LES OBJETS JAVA CARD

Pour représenter, stocker, et manipuler des données le JCRC et les applets créent des objets. Sur la plate-forme Java Card ces objets obéissent aux règles de programmation du langage Java :

- tous les objets sur la plate-forme Java Card sont des instances, créées dans le tas (*i.e.* en anglais *heap*), de

classes ou de tableaux, qui ont la même classe racine `java.lang.Object` ;

- les champs d’un nouvel objet ou les éléments d’un nouveau tableau sont initialisés à leurs valeurs par défaut (0, `null` or `false`) sauf si ils sont initialisés à une autre valeur par le constructeur.

La technologie Java Card autorise les objets *persistants* et les objets *transients*. Pourtant, les concepts d’objets persistants et transients et les mécanismes associés sur la plate-forme Java Card ne sont pas les mêmes que ceux de la plate-forme Java standard.

5 LA NATURE DES OBJETS JAVA CARD

Dans la technologie Java Card, le lieu de stockage des valeurs d’un objet dépend de sa nature.

5.1 Les objets persistants

Les valeurs des champs d’un objet persistant sont persistantes et sont donc stockées en mémoire persistante. Les spécifications Java Card précisent aussi qu’un objet persistant doit être créé :

- lorsque la méthode `Applet.register` est appelée ;
- lorsque une référence à l’objet est stockée dans un champ d’un objet persistant ou dans celui d’une classe.

5.1.1 Caractéristiques d’un objet persistant

Un objet persistant est un objet qui a été créé par l’opérateur `new` (*i.e.* par un des bytecode `new`, `anewarray` ou `newarray`) avant de devenir persistant.

Un tel objet conserve ses valeurs au fur et à mesure des sessions avec le lecteur.

Chaque mise à jour d’un champ d’un objet persistant est atomique (*cf.* Section 6.1) et un objet persistant prend part aux transactions (*cf.* Section 6.2).

Un objet persistant peut être référencé par un champ d’un objet transient.

5.2 Les objets transients

La notion de *transient* permet d’avoir des objets dont les champs ont un contenu temporaire. Il existe deux types d’objets transients, nommés `CLEAR_ON_RESET` et `CLEAR_ON_DESELECT`. Chaque type est associé à un événement qui lors de son occurrence déclenche la réinitialisation des champs de l’objet.

Les objets transients `CLEAR_ON_RESET` sont utilisés pour conserver des données qui doivent exister tout au long d’une session avec le lecteur et donc au delà des sessions de l’applet. Ces objets sont réinitialisés lorsque l’évènement `CLEAR_ON_RESET` se produit. Il est provoqué par le redémarrage du périphérique explicitement ou implicitement (comme à la suite d’une coupure de l’alimentation).

Les objets transients `CLEAR_ON_DESELECT` sont utilisés pour conserver des données qui doivent exister seulement pendant la durée d’une session de l’applet (*i.e.* le temps où une applet est sélectionnée). Ces objets sont réinitialisés lorsque l’évènement `CLEAR_ON_DESELECT` se produit, *i.e.* lors de la sélection d’une autre applet ou de l’occurrence d’un évènement `CLEAR_ON_RESET`. En effet, un

redémarrage du périphérique implique en particulier la désélection implicite de l’applet sélectionnée. Les objets transients `CLEAR_ON_DESELECT` sont donc aussi des objets transients `CLEAR_ON_RESET`.

Il est important de noter que ces événements affectent seulement le contenu des champs de l’objet et aucune-ment son entête dont les informations persisteront tout au long de la vie de l’objet. Par exemple dans le cas des tableaux transients qui sont les seuls types objets transients spécifiés jusqu’à maintenant dans Java Card, l’attribut de taille du tableau n’est pas remis à jour lors de l’occurrence d’un événement `CLEAR_ON_DESELECT` ou `CLEAR_ON_RESET`. En effet le contenu de ce tableau transient a été réinitialisé mais occupe toujours l’espace mémoire qui lui avait été alloué. On remarquera donc que le terme anglais *transient*, qui veut dire « temporaire » ou « transitoire », a été très mal choisi pour exprimer la nature réelle de ces objets.

5.2.1 Caractéristiques d’un objet transient

Un objet transient est créé par l’invocation de méthodes de la classe `JCSys-tem` :

- `makeTransientBooleanArray` ;
- `makeTransientByteArray` ;
- `makeTransientShortArray` ;
- `makeTransientObjectArray`.

Un objet transient ne conserve pas ses valeurs à travers les sessions avec le lecteur. Ses champs sont remis à leurs valeurs par défaut (0, `false`, `null`) lors de l’occurrence de certains événements.

Chaque mise à jour d’un champ d’un objet transient est non atomique et un objet transient ne participe pas aux transactions.

Un champ d’un objet transient peut référencer un objet persistant.

Les objets de nature transiente sont utilisés en raison de leur rapidité et leur sécurité (*cf.* Section 8.2.2) – p.ex. pour le stockage de clés cryptographiques.

On notera aussi qu’un tableau transient d’objets est en fait un tableau de références désignant des objets qui peuvent être de nature transiente ou persistante. Ainsi sur l’occurrence de l’évènement associé au tableau les valeurs du tableau sont réinitialisées mais les objets précédemment référencés continuent d’exister en mémoire.

6 ATOMICITÉ ET TRANSACTION

Afin de garantir l’intégrité des données lors de la mise à jour des objets persistants, les notions d’atomicité et de transaction ont été introduites dans l’environnement Java Card.

6.1 Atomicité

Une opération est atomique si elle est soit totalement exécutée soit pas du tout.

L’interpréteur Java Card doit assurer que la mise à jour des champs des objets persistants est atomique afin de palier à tout incident pouvant mettre en péril l’intégrité

des données modifiées (perte d'alimentation, redémarrage de la puce par un détecteur de rayonnement lumineux anormaux [Skorobogatov, 2002], injection de fautes [Bar-El, 2004, Giraud, 2004], etc.). Ainsi lors d'une opération d'affectation, le champ (d'instance ou de classe) ou l'élément de tableau concerné pourra soit prendre la nouvelle valeur ou soit conserver la valeur précédente. En aucun cas une autre valeur n'est admissible.

Les APIs permettent aussi la mise à jour atomique d'une portion d'un tableau persistant grâce à la méthode `Util.arrayCopy`. Les applets n'ayant pas ces exigences d'intégrité utiliseront la méthode `Util.arrayCopyNonAtomic`, plus rapide, puisque ne nécessitant pas la sauvegarde des anciennes valeurs.

6.2 Transaction

Une transaction représente un bloc d'opérations qui doit être effectué complètement ou pas du tout. Le principe est le même que celui qui existe dans le domaine des bases de données.

En Java Card, le déclenchement, la terminaison ou l'annulation des transactions se fait via des méthodes des APIs (*i.e.* respectivement `beginTransaction`, `commitTransaction` et `abortTransaction` de la classe `JCSystem`). Au niveau applicatif, un seul niveau de transaction est autorisé.

Pour restaurer les anciennes données en cas d'échec de la transaction (ou de la mise à jour atomique) ou en cas d'annulation de la transaction via la méthode `abortTransaction`, la Java Card doit posséder un mécanisme de recouvrement (*i.e.* en anglais *rollback*) des données. La restauration des données implique la sauvegarde dans le buffer de transaction des anciennes valeurs des champs des objets persistants modifiés. Sa taille est dépendante de l'implantation et conditionne le nombre de mises à jour possibles au sein d'une transaction. La sauvegarde d'une ancienne donnée correspond à la sauvegarde de sa valeur mais aussi à la sauvegarde d'informations supplémentaires [Oestreicher, 1999a] telles que sa position mémoire, son type, etc. La taille et la nature des informations sauvegardées sont dépendantes de l'implantation. Ainsi un programme pourra fonctionner parfaitement sur une implantation et mal se comporter sur une autre dans les conditions limites.

Il existe de nombreux problèmes liés à l'instanciation d'objets à l'intérieur des transactions annulées [Oestreicher, 1999a].

6.3 Perte d'alimentation et reset

Lors d'une perte d'alimentation ou d'un reset, toutes les données en mémoire volatile sont perdues. À la remise sous tension le JCRE doit remettre la carte dans un état cohérent. En particulier il doit annuler les éventuelles transactions en cours avant de pouvoir émettre son ATR² et d'accepter une commande.

²Réponse à la mise sous tension de la carte.

7 GESTION DE LA MÉMOIRE

7.1 Allocation

L'interprète alloue :

- les champs statiques des classes au chargement des fichiers CAP ;
- la représentation des objets dans le tas au travers :
 - des bytecodes `new` pour les objets, `newarray` pour les tableaux classiques de type simple et `anewarray` pour les tableaux classiques de type référence,
 - des méthodes des APIs pour les tableaux transients (p.ex. `makeTransientByteArray`, etc.).

7.2 Initialisation

L'initialisation des champs statiques se fait après le chargement d'un paquetage avec les valeurs que l'on trouve dans le composant `StaticField` du fichier CAP. L'initialisation des champs statiques est soumise à quelques contraintes dans la technologie Java Card. En effet, les champs statiques d'un paquetage d'applet peuvent seulement être initialisés à des valeurs constantes de type primitif ou à des tableaux de constantes de type primitif alors que les champs statiques d'un paquetage de bibliothèque utilisateur peuvent seulement être initialisés à des valeurs constantes de type primitif. Seuls les champs statiques déclarés dans une classe d'un paquetage peuvent être initialisés par ce paquetage.

Les bytecodes de création et les méthodes des APIs devraient initialiser les champs des objets à leur valeur par défaut.

7.3 Désallocation

Une nouvelle fois, les désallocations se font soit au niveau interprète soit au niveau JCRE. La destruction de différentes structures de données peut se faire au niveau des octets dans les mémoires physiques de différentes manières :

- par passage des octets aux valeurs naturelles de la mémoire volatile lorsqu'elle n'est plus alimentée (*i.e.* `0x00` ou `0xFF` suivant le type de la mémoire) ;
- par passage des octets à des valeurs aléatoires afin d'éviter des phénomènes de rémanence [Gutmann, 2001].

En Java Card, comme en Java, il n'y a pas de mot-clé pour la libération explicite de la mémoire. Comme la spécification ne rend pas obligatoire la présence des mécanismes de ramasse-miette, il n'y a pas de libération de la mémoire allouée qui n'est plus utilisée, sauf lors de l'effacement d'une applet et dans certains autres cas comme cela est implicitement suggéré dans les spécifications (p.ex. lors de l'échec d'une installation). Ainsi dès qu'un objet ne sera plus référencé, cet objet ne sera plus jamais accessible et l'espace mémoire qu'il occupe sera perdu. Cependant depuis sa version 2.2, Java Card prévoit la présence optionnelle d'un mécanisme de ramasse-miette avec déclenchement à la demande de l'applet via la méthode `JCSystem.requestObjectDeletion()`. On remarquera que le comportement d'une applet dépendra

donc de la Java Card sur laquelle elle s'exécute, s'éloignant un peu plus de la philosophie Java : « Write once, run anywhere ». Certains constructeurs ont aussi implanté des mécanismes de ramasse-miette avec un déclenchement à la demande en provenance de l'application client au travers d'une commande envoyée à la carte. De toute façon on notera qu'un mécanisme de ramasse-miette nuit aux performances de la carte à cause de la lenteur des écritures en EEPROM mais aussi à sa durée de vie, une fois encore à cause de l'EEPROM et de son nombre de cycles d'écriture limité.

8 INTERPRÉTATIONS DES SPÉCIFICATIONS

Les spécifications Java Card ne précisent pas la localisation des structures de données dans les mémoires physiques. En effet, Sun microsystems a fait le choix de laisser toute liberté au développeur, permettant ainsi d'adapter la technologie Java Card à plusieurs types de périphériques (*i.e.* cartes à puce, iButtons, etc.). Seulement, malgré la précision des spécifications, des zones d'ombre, voire mêmes des contradictions, sont présentes laissant une large place à l'interprétation. Or cela risque d'aboutir à des implantations qui n'ont pas du tout le même niveau de sécurité, ni les mêmes fonctionnalités.

Dans la suite plusieurs interprétations possibles de cette partie des spécifications vont être détaillées et comparées. Les périphériques considérés seront naturellement les cartes à puce, cibles privilégiées de la technologie Java Card.

8.1 Le problème de la définition du tas

Lorsqu'on veut étudier la description de la mémoire dans les spécifications Java Card, la première difficulté consiste à trouver des définitions cohérentes et homogènes des différents termes tout au long des documents. Ainsi les spécifications définissent le tas tantôt comme une zone de mémoire libre utilisable par le programme et tantôt comme un ensemble d'objets stockés dans une mémoire persistante. Pour éviter toute confusion, nous emploierons la définition du tas largement répandue qui consiste à considérer le tas comme la zone mémoire qui comprend l'ensemble des objets alloués et de la mémoire libre. Ainsi défini le tas aura donc une taille fixe tout au long de la vie de la Java Card et seules les tailles totales de l'espace alloué aux objets et de la mémoire libre évolueront proportionnellement et en sens inverse dans le temps.

Bien que les spécifications ne fassent pas état de la notion de tas statique, nous définirons le tas statique comme l'espace contenant l'ensemble des champs statiques des paquetages présents sur la carte. Évidemment il aurait été possible de considérer le tas statique et le tas comme une seule et même entité, mais des recherches [Barthe, 2001] sur la formalisation de la JCVM introduisaient cette notion que nous avons souhaité conserver par soucis de cohérence. Cet espace contient donc des structures de données Java Card (*i.e.* de type primitif et référence) mais en aucun cas les objets Java Card désignés par les champs de type référence. Par essence les données et les structures

du tas statique doivent exister au travers des sessions avec le lecteur. Aussi le tas statique doit être localisé dans une mémoire persistante, et donc en EEPROM dans le cas des cartes à puce.

8.2 Le problème du contenu du tas

8.2.1 Le tas et les objets persistants

Tout naturellement les objets persistants sont localisés dans l'espace persistant. Cette partie du tas devant elle même être localisée dans une mémoire persistante telle que l'EEPROM.

8.2.2 Le tas et les objets transients

Les spécifications sont très précises concernant la localisation des objets transients. Les informations sur leur structure sont localisées en mémoire persistante donc dans l'espace persistant alors que les valeurs de leurs champs sont localisées dans l'espace transient. Les spécifications déconseillent fortement de localiser l'espace transient dans une mémoire persistante. Il devrait donc être localisé pour les cartes à puce en RAM. L'écriture dans les champs des objets transients ne pose pas de problème de performance car le cycle d'écriture de la RAM est beaucoup plus court que celui de l'EEPROM. Les propriétés de ces objets (*i.e.* rapidité et sécurité) sont issues des propriétés de la RAM.

8.2.3 La notion de pré-persistence

La première imprécision des spécifications Java Card concerne la localisation des structures des objets Java Card lors de leur création. En effet, les spécifications du JCRE (*cf.* [Sun, 2003] Section 2) énoncent clairement que le développeur *rendra* les objets persistants lorsque la méthode `Applet.register` est invoquée ou lorsqu'une référence à l'objet est stockée dans un champ d'un objet persistant ou celui d'une classe. Par ailleurs elles précisent aussi qu'un objet est créé depuis le tas lors de :

- l'exécution d'un bytecode `new`, `anewarray` et `newarray` ;
- l'invocation d'une méthode de l'API `makeTransientXXXArray`.

Aussi quelle est la nature des objets entre le moment de leur création et celui de leur affectation à un champ d'un objet persistant ?

Nous avons choisi de désigner par *espace pré-persistant* l'espace mémoire où sont créés les objets après l'exécution d'un bytecode `new`, `anewarray` ou `newarray`. Mais attention, le nom de cet espace n'implique pas que les objets qu'il contient deviendront forcément persistants.

Toutefois dans le glossaire des spécifications 2.2.1 du JCRE se trouve une définition des objets persistants qui décrit les objets comme étant persistants par défaut. Or cette définition va à l'encontre de l'exigence énoncée par la spécification. Quelle partie des spécifications faut-il implanter ?

Ainsi le développeur de plate-forme Java Card doit choisir entre deux implantations quant au contenu du tas (*cf.* FIG. 2) :

- une allouant directement les objets créés dans l'espace

- persistant et se conformant ainsi au glossaire ;
- l’autre possédant la notion d’espace pré-persistant introduit ci-dessus et donc appliquant l’exigence.

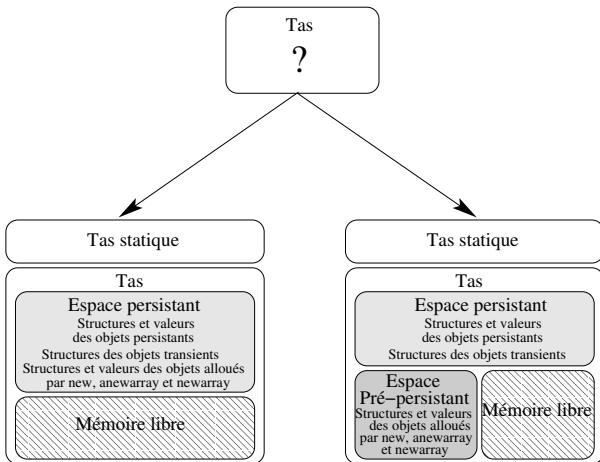


FIG. 2 – Le problème du contenu du tas.

Quels sont les avantages et les inconvénients de ces deux solutions ? La solution sans espace pré-persistant offre une facilité d’implantation puisqu’il suffit d’allouer simplement tous les objets dans l’espace persistant sur les bytecodes de création ou sur les méthodes des APIs. La solution avec l’espace pré-persistant offre plusieurs avantages :

- la rapidité lors des opérations d’écritures puisque seuls les champs des objets persistants sont touchés par l’atomicité et les transactions ;
- une implantation facile d’un mécanisme de ramasse-miette dans l’espace pré-persistant en libérant cet espace à la demande ou à chaque redémarrage du périphérique ;
- la possibilité de l’utiliser pour créer un Environnement Transient [Oestreicher, 1999b] qui offre au langage un plus grand pouvoir d’expression et pour résoudre des problèmes liés à la création d’objets dans une transaction annulée [Oestreicher, 1999a].

Son inconvénient principal réside dans l’implantation des mécanismes de recopie des objets de l’espace pré-persistant dans l’espace persistant. En effet, ces deux espaces peuvent être localisés dans des mémoires de types différents comme nous le verrons plus loin. De plus la recopie doit être réalisée au sein d’une transaction et nécessite la mise en oeuvre d’algorithmes de parcours d’objets afin de déterminer les objets à rendre persistants. Par exemple dans les cas où des objets pré-persistants se référencent entre eux, tous devraient devenir persistants. L’opération d’affectation à un champ d’un objet persistant ou à un champ de classe serait donc contaminant. L’imprécision relevée plus haut implique également qu’un objet transient dont la référence serait affectée à un champ d’un objet persistant ou celui d’une classe deviendrait persistant. Ainsi seuls les objets transients référencés par des variables locales, par la pile d’opérandes ou par des objets non persistants pourraient garder leur transience. Dans les autres cas de référencement ces objets

transients deviendraient des objets persistants et ils devraient donc les valeurs de leurs champs migrer de l’espace transient vers l’espace persistant. Un autre avantage de la solution utilisant l’espace pré-persistant est donc que les objets de cet espace pourraient eux aussi référencer un objet transient sans en changer sa nature.

Pour résumer, un objet pré-persistant :

- serait créé lors de l’exécution d’un bytecode `new`, `anewarray`, `newarray` ou de l’invocation d’une méthode de l’API `makeTransientXXXArray` ;
- ne participerait pas aux mécanismes d’atomicité et de transactions ;
- pourrait référencer des objets persistants, transients et pré-persistant ;
- pourrait être référencé par des objets transients et pré-persistant mais pas par des objets persistants au risque de devenir persistant lui-même.

8.3 Le problème de la localisation du tas

La seconde imprécision des spécifications concerne toujours le tas. En effet, les versions antérieures aux spécifications 2.2 ne donnaient aucune définition de la notion de tas et, aujourd’hui encore malgré les précisions, la liberté est laissée au développeur du JCRE de choisir l’étendue et la localisation du tas.

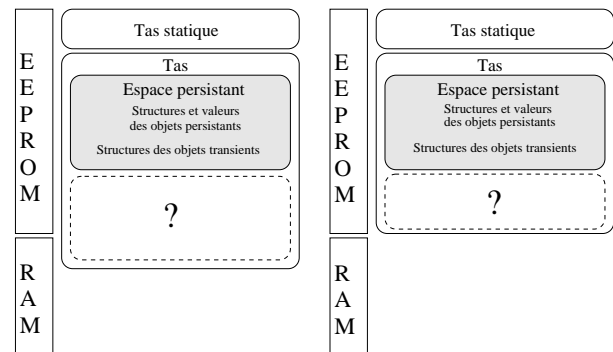


FIG. 3 – Le problème de l’étendue du tas.

Le tas doit-il être localisé (cf. FIG. 3) seulement en EEPROM, ou bien est-il localisé en EEPROM et en RAM ? La seule certitude sur le tas est qu’une partie se situe en EEPROM puisqu’il doit contenir l’espace de stockage des objets persistants (*i.e.* structures et valeurs) et celui des structures des objets transients.

8.4 Le problème global du tas

Or de ces propriétés sur le tas peuvent être déduites des propriétés sur la nature des objets au moment de leur création. Il faut donc étudier le problème global qui résulte de la combinaison des deux imprécisions des spécifications (cf. FIG. 4).

En réponse au problème global, nous avons dégagé quatre solutions d’implantation différentes du tas avec des avantages et des inconvénients pour chacune.

La solution n°1 (cf. FIG. 5) est celle retenue dans la plupart des implantations Java Card. Son principal avantage

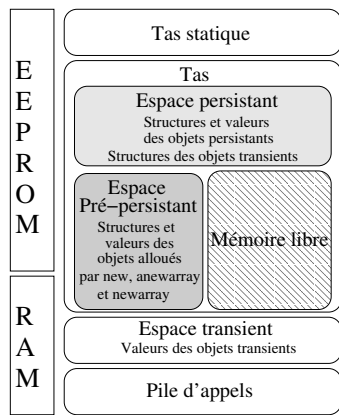


FIG. 4 – Le problème global de l'étendue et du contenu du tas.

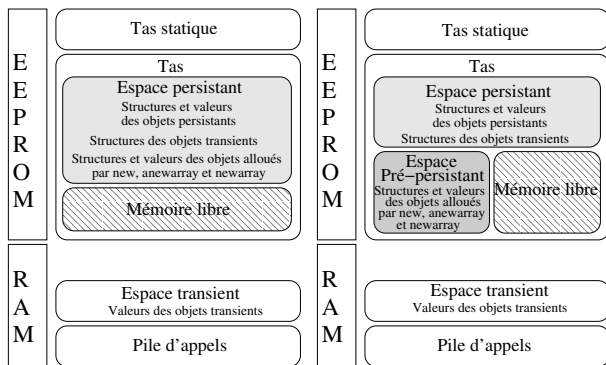


FIG. 5 – La solution n°1

FIG. 6 – La solution n°2.

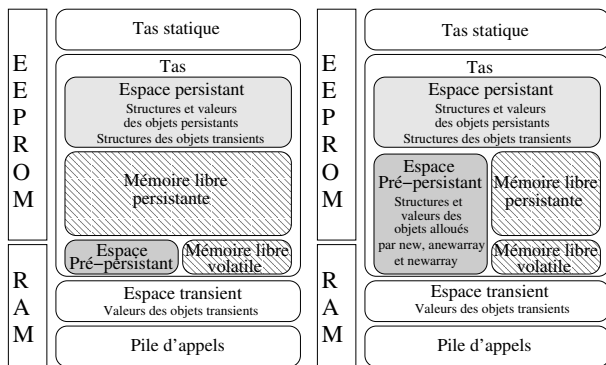


FIG. 7 – La solution n°3.

FIG. 8 – La solution n°4.

est sa facilité d'implantation au niveau de la gestion de l'allocation des objets. Ses inconvénients sont :

- une sémantique différente de celle du langage Java pour l'allocation des objets puisque les objets sont alloués dans un tas en mémoire persistante (*i.e.* EEPROM) et non en mémoire volatile comme en Java (*i.e.* RAM du PC) ;
- une certaine lenteur à l'exécution causée par les mécanismes d'atomicité et de transaction qui sont plus lent en EEPROM qu'en RAM.

La solution n°2 (*cf.* FIG. 6) est un peu plus difficile à

implanter en raison de la gestion du passage des objets de l'espace pré-persistant à l'espace persistant lors des opérations définies dans les spécifications du JCRE (*cf.* [Sun, 2003] Section 2). En revanche par rapport à la solution n°1 le gain en rapidité est non négligeable en raison de l'absence des mécanismes d'atomicité et de transaction pour l'espace pré-persistant. L'implantation d'un mécanisme de ramasse-miette partiel peut être assez facile, puisqu'il suffit à chaque redémarrage du périphérique de libérer l'espace pré-persistant.

La solution n°3 (*cf.* FIG. 7) est sûrement la plus performante en terme de sécurité et de rapidité. La rapidité est accrue grâce à la localisation de l'espace pré-persistant dans la RAM. Mais il faut nuancer ce gain par la faible taille de la RAM. La sécurité est accrue par la possibilité de créer des objets dans la RAM. Ainsi les tableaux transients d'objets ont un réel intérêt puisque contrairement à la solution n°1 les objets créés peuvent être dans une mémoire volatile et seront donc détruit physiquement entre deux sessions avec le lecteur (*cf.* Section 5.2.1). Cette propriété obtenue avec cette implantation semble plus naturelle et conforme à l'esprit qu'on peut se faire d'un tableau transient d'objets. Un dernier avantage de cette solution est le mécanisme de ramasse-miette naturel de la RAM à chaque mise hors-tension reléguant l'implantation du ramasse-miette à un simple nettoyage de la table d'allocation de la RAM. Effectivement, rappelons que les objets transients ont vu leur contenu disparaître mais qu'ils existent toujours (leur structure est persistante) et ils requièrent donc toujours de la mémoire RAM qu'il faut marquer comme déjà occupée dans la table d'allocation. Comme nous l'avons expliqué précédemment le principal problème réside dans la recopie des objets de l'espace pré-persistant en RAM dans l'espace persistant qui est lui en EEPROM.

La solution n°4 (*cf.* FIG. 8) offre un bon compromis entre la rapidité et la taille des ressources mémoires. En effet la RAM de l'espace pré-persistant peut être utilisée pour créer les objets par défaut et en cas de mémoire libre volatile insuffisante l'objet sera alloué dans l'EEPROM. Son inconvénient est la diminution de la rapidité d'exécution et de la sécurité quand l'espace pré-persistant en RAM est totalement occupé. Les autres propriétés de cette solution sont les mêmes que celles des solutions n°2 et n°3. On notera qu'il est possible d'appliquer différentes politiques d'allocation pour ce modèle.

9 TRAVAUX CONNEXES

Nos propositions se différencient de celles faites en 1999 par IBM et Schlumberger [Oestreicher, 1999b] sur plusieurs points. Tout d'abord leurs travaux se plaçaient dans le cadre la solution n°1, *i.e.* sans espace pré-persistant et avec le tas en EEPROM. Ensuite ils s'intéressaient exclusivement à expliquer les différentes solutions possibles pour définir la transience. C'est dans ce cadre qu'ils ont introduit la notion très séduisante d'Environnement Transient explicite. Ce concept semblait s'intégrer si bien dans la philosophie du langage Java Card qu'on pourrait se demander pourquoi le Java Card Forum et Sun micro-

systems n'ont pas mis en oeuvre leur proposition. En fait ils n'ont probablement pas voulu changer la sémantique des spécifications. Quoiqu'il en soit si nos travaux se concentrent principalement sur les problèmes issus de la nature persistante des objets, ils sont aussi complémentaires quant à la notion de transience. En effet la notion de pré-persistence que nous avons introduit permet elle aussi la mise en oeuvre des concepts d'Environnements Transients implicite et/ou explicite.

10 CONCLUSION ET PERSPECTIVES

Dans cet article nous avons :

- identifié deux points ambigus dans les spécifications Java Card.
- introduit la notion de *pré-persistence*.
- décrit les quatre modèles mémoires possibles, leurs avantages et leurs inconvénients. Nous aurions également souhaité détailler dans la section 8.4 les problèmes sémantiques de chacun des modèles et les illustrer par des exemples. Faute de place cet article décrit les possibilités d'implantation et leur étude détaillée fera l'objet d'une publication dédiée.

S'il est vrai que parmi les différents modèles de mémoire évoqués, il semble que tous les développeurs aient choisi la solution basée sur un tas localisé en EEPROM et sans espace pré-persistant, il nous paraîtrait souhaitable que le Java Card Forum³ réexamine la gestion de la mémoire afin de garantir une vraie interopérabilité entre toutes les implantations. La prochaine étape consiste à implanter et tester ces différents modèles sur notre plate-forme Jca-tools. Ceci nous permettra d'étudier plus avant l'impact effectif des différentes stratégies que nous avons décrites sur des cas réels.

BIBLIOGRAPHIE

- [Arnold, 2000] ARNOLD K., GOSLING J., HOLMES D., "The Java programming language (3rd Edition)". Addison-Wesley, ISBN 0201704331 (2000).
- [Atkinson, 1995] ATKINSON M. P., MORRISON R., "Orthogonally persistent object systems". The VLDB Journal, 4(3), 1995.
- [Bar-El, 2004] BAR-EL H., CHOUKRI H., NACCACHE D., TUNSTALL M., WHELAN C., "The Sorcerer's Apprentice Guide to Fault Attacks". In Proc. of Workshop on Fault Detection and Tolerance in Cryptography, Italy, 2004.
- [Barthe, 2001] BARTHE G., DUFAY G., JAKUBIEC L., SERPETTE B., MELO DE SOUSA S., "A Formal Executable Semantics of the JavaCard Platform". In Proc. of the 10th European Symposium on Programming Languages and Systems (ESOP 2001), Genova, Italy, April 2-6, 2001.
- [Chaumette, 2003] CHAUMETTE S., HATCHONDO I., SAUVERON D., "JCAT : An environment for attack and test on Java Card". In Proc. of the International Conference on Computer, Communication and Control Technologies (CCCT03), Orlando, Florida, USA, July 31, August 1-2, 2003.
- [Chen, 2000] CHEN Z., "Java Card™ Technology for Smart Cards". Addison-Wesley, ISBN 0201703297 (2000).
- [Giraud, 2004] GIRAUD C., THIEBEAULD H., "A survey on fault attacks". In Proc. of CARDIS'04, Smart Card Research and Advanced Applications VI, Kluwer academic publisher, pp. 159-176, Toulouse, France, August 24-26, 2004.
- [Gosling, 2000] GOSLING J., JOY B., STEELE G., BRACHA G., "The Java Language Specification (2nd Edition)". Addison-Wesley, ISBN 0201310082 (2000).
- [Gutmann, 2001] GUTMANN P., "Data Remanence in Semiconductor Devices". In Proc. of 10th Usenix Security Symposium, Washington, D.C., USA, August 13-17, 2001.
- [Oestreicher, 1999a] OESTREICHER M., "Transactions in Java Card". In Proc. of 15th Annual Computer Security Applications Conference (ACSAC), Phoenix, Arizona, USA, 1999.
- [Oestreicher, 1999b] OESTREICHER M., KRISHNA K., "Object lifetimes in Java Card". In Proc. of Usenix Workshop on Smartcard Technology (Smartcard'99), Chicago, Illinois, USA, May 10-11, 1999.
- [Sauveron, 2001] SAUVERON D., "La technologie Java Card : Présentation de la carte à puce. La Java Card". Rapport interne du LaBRI RR-1259-01.
- [Sun, 2003] Sun microsystems, "Java Card™ specifications 2.2.1". <http://java.sun.com/products/javacard/>
- [Skorobogatov, 2002] SKOROBOGATOV S., ANDERSON R., "Optical Fault Induction Attacks". In Proc. of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002), San Francisco Bay (Redwood City), USA, August 13-15, 2002.

³Organisme qui soumet des propositions d'évolution des spécifications Java Card à Sun microsystems