N° d'ordre : 3724

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Achraf KARRAY**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Conception, mise en œuvre et validation d'un environnement logiciel pour le calcul sécurisé sur une grille de cartes à puce de type Java

---

Soutenue le : 10 décembre 2008

Après avis des rapporteurs :

Didier Donsez ...	Professeur
Pierre Paradinas	Professeur

Devant la commission d'examen composée de :

Éric Sopena	Professeur	Président
Didier Donsez ...	Directeur de Recherche	Rapporteur
Pierre Paradinas	Professeur	Rapporteur
Damien Sauveron	Maître de Conférences	Examinateur
Serge Chaumette	Professeur	Directeur de thèse

– 2008 –

Déclaration

Ce doctorat a été réalisé sous la direction de Serge Chaumette.

Je déclare que les travaux présentés dans cette thèse sont le résultat de recherches originales menées en collaboration avec d'autres dans le cadre de mon inscription à l'Université Bordeaux 1 afin d'obtenir le titre de Docteur en Informatique. Ils n'ont été présentés dans aucune autre université ou établissement d'éducation en vue d'obtenir un autre diplôme ou une autre distinction.

Table des matières

Liste des tableaux

Table des figures

Table des Listings

Chapitre 1

Introduction

1.1 Contexte

Nombreuses sont aujourd'hui les applications qui nécessitent une puissance de calcul importante. C'est par exemple le cas de la prévision météo ou de la modélisation de protéines qui requièrent de plus en plus de cycles processeurs. Parallèlement, et grâce aux progrès technologiques, les performances des ressources matérielles ne cessent de croître. La puissance de calcul fournie n'est néanmoins pas toujours suffisante pour répondre aux attentes des utilisateurs.

Des architectures de type *cluster* ont donc été conçues pour fournir une puissance de calcul importante. Un *cluster*, dit aussi grappe, est l'interconnexion de plusieurs machines appelées nœuds. Le coût d'une telle architecture est très élevé et n'est donc pas à la portée de tous. Une autre solution est alors l'utilisation de grilles [13, 58], les grilles sont une extension du concept de cluster. Dans un cluster les différents nœuds sont rassemblés en *racks* dans un espace limité, alors qu'une grille est une architecture largement distribuée reliant un ensemble de ressources (clusters, super-calculateurs, serveurs, etc.). Une grille peut être définie comme une infrastructure matérielle et logicielle permettant de rendre accessibles les ressources disponibles sur différents sites. Des environnements logiciels tels que Globus [55] ou Proactive [118] ont été développés afin de permettre l'exploitation d'un ensemble de machines pour réaliser des calculs intensifs et nécessitant donc une importante puissance CPU. Une autre des formes d'utilisation des grilles est le calcul opportuniste qui consiste à utiliser les cycles de calcul des machines connectées sur le réseau et qui sont non utilisées. Les propriétaires d'applications exécutent donc leurs programmes sur des machines qui ne leur appartiennent pas. Il existe aujourd'hui plusieurs utilisations de ce type d'architecture. À titre d'exemple, citons : le projet SETI@home [131] dont le but est de chercher une forme d'intelligence extra-terrestre en analysant les signaux reçus de l'espace ; le projet Decrypton [41] dont l'objectif est la modélisation des protéines afin d'accélérer la recherche en génomique et protéomique ; le challenge lancé par les *RSA Laboratories* [127] pour casser des clés cryptographiques.

Dans ces configurations, les propriétaires des codes et des matériels sont distincts, et des problèmes de confiance et de sécurité se posent donc. Il faut garantir à la fois :

- la sécurité du support d'exécution. Il s'agit de la sécurité du système, des services et applications de la machine hôte (atteinte aux données privées, programme hostile de type virus ou cheval de troie, etc.) ;
- la sécurité de l'application. Ceci concerne notamment la confidentialité du code, l'intégrité de l'exécution du programme, la protection contre toute forme d'altération des résultats fournis, etc.

Une confiance mutuelle doit donc être établie entre le fournisseur du code et le propriétaire de la machine qui l'exécute, chacun pouvant être dangereux pour l'autre. Dans ce contexte, un utilisateur est obligé de faire totalement confiance au propriétaire du matériel sur lequel il exécute son code. Rien ne le protège d'opérations malveillantes sur son programme qui pourraient découler d'un accès physique à l'unité de calcul : enregistrement de données sensibles, perturbation du calcul et/ou des

résultats produits. Le propriétaire du matériel quant à lui n'a pas de moyen de vérifier l'innocuité des programmes s'exécutant sur sa machine. L'application pourrait contenir un code malveillant visant à accéder à ses données privées et/ou à endommager son système (tels que virus, cheval de troie, bombe logique, etc.).

1.2 Problématique

Mettre son matériel à la disposition de tiers pour exécuter des applications comporte donc un potentiel danger. La question qui se pose est de faire confiance ou pas au propriétaire du code ? En effet, il est tout à fait possible que l'application cache un virus ou un cheval de troie ou qu'elle contienne un code malicieux visant à récupérer des données sensibles. De son côté, l'utilisateur doit faire confiance au propriétaire du matériel. Les problèmes de sécurité qui se posent sont les suivants :

- **confidentialité du code versus sécurité du support d'exécution** : afin de pouvoir vérifier l'innocuité du code à exécuter, le propriétaire du matériel peut souhaiter que l'application soit en *open source*, ou au moins avoir accès aux sources. Cette possibilité n'est, malheureusement, pas toujours envisageable. Le propriétaire du code peut, pour diverses raisons, ne pas vouloir dévoiler à des tiers les détails d'implémentation de son application. Il ne délivre alors aux différentes ressources de calcul qu'une version binaire de son programme. Ceci n'est pas suffisant pour garantir la confidentialité de l'application car le propriétaire du matériel, peut par exemple très facilement tracer l'exécution de l'application et connaître son code en faisant une simple copie de la mémoire.
  En résumé, on peut dire que la confidentialité d'une application est une propriété très difficile voire impossible à garantir. Ceci est encore plus compliqué lorsqu'il s'agit en plus de vérifier l'innocuité de l'application vis à vis du support d'exécution.
- **intégrité de l'exécution de l'application** : outre la confidentialité, le propriétaire du code cherche à garantir l'intégrité de l'exécution de son application. Autrement dit, il doit s'assurer que les résultats fournis ne sont pas altérés ou falsifiés. En effet, le propriétaire du matériel pourrait agir sur l'application ou perturber son exécution, ce qui pourrait amener à la génération de résultats erronés.
- **sécurité des échanges** : la falsification des résultats peut aussi intervenir à posteriori de l'exécution, pendant les échanges intervenant entre les différents processus du calcul et/ou d'administration. Par exemple, le propriétaire du matériel, ou même un intrus, peut altérer les messages envoyés par l'application. Il est donc nécessaire de mettre en place des mécanismes de sécurité afin de protéger les communications entre l'utilisateur et son application.

1.3 Notre solution

Une des solutions possibles aux problèmes évoqués précédemment est l'utilisation de matériels sécurisés tels que des cartes à puce [30, 32, 53]. La carte à puce présente en effet des caractéristiques de sécurité qui ne sont pas fournies par les autres systèmes ; elle est considérée aujourd'hui comme le système informatique le plus sûr [145]. Cette sécurité se traduit par des mécanismes d'isolation qui offrent une séparation complète entre les différentes applications. Chacune s'exécute dans son propre environnement cloisonné à partir duquel elle ne pourra pas mettre en péril le système ni affecter les autres programmes. Ainsi, l'exécution d'un code malicieux n'a aucun effet sur la carte ou les applications embarquées. Au niveau matériel, la carte à puce bénéficie de protections physiques qui rendent presque impossible la perturbation des applications qu'elle exécute, et par conséquent la génération de résultats erronés. Les détecteurs de conditions de fonctionnement hors norme qui sont intégrés permettent de se prémunir contre les attaques visant à accéder aux données et aux applications qui y sont hébergées. De plus, les différents mécanismes de protection mis en œuvre peuvent être évalués et certifiés par des organismes habilités tels que les CESTI (Centres d'Évaluation de la Sécurité des Technologies de l'Information) en France [40].

En se basant sur une architecture de type cluster, nous définissons une infrastructure logicielle pour le calcul distribué sécurisé sur grille de cartes à puce[1] permettant d'exploiter les différents cartes mises à disposition.
Bien évidemment, il ne s'agit pas de réaliser des calculs intensifs vues les limitations en ressources des cartes, mais de réaliser une preuve de concept et de traiter des cas d'utilisation et des applications dans lesquelles l'aspect sécurité prime sur l'aspect calcul.

1.4 Objectifs

L'objectif que nous poursuivons est de définir un modèle référentiel pour les grilles de cartes à puce en spécifiant une architecture logicielle pour ce type de grille. Il s'agit également de réaliser la conception, la mise en œuvre et la validation d'un environnement logiciel associé permettant d'effectuer des calculs distribués sécurisés. Pour atteindre ces objectifs, il faut :

- proposer une architecture générique pour les grilles de cartes à puce. Le fait de disposer d'une architecture bien définie sera d'une grande utilité pour expliquer certains concepts fondamentaux de ce type de grille, découlant de la spécificité et des particularités des cartes à puce par rapport à d'autres matériels ;
- concevoir et mettre en place un support d'exécution (un intergiciel) pour le calcul sécurisé. Il s'agit d'assurer la sécurité du matériel, du code de l'application et de son exécution, et des échanges des données ;
- faciliter l'interaction avec les cartes à puce. Ceci revient à définir une méthode d'accès de haut niveau permettant d'offrir une abstraction vis à vis du protocole de communication traditionnel, qui est relativement primitif et très contraignant ;
- proposer des outils d'assistance. Afin d'aider les utilisateurs de la grille à exploiter et à gérer les ressources matérielles et à exécuter leurs applications, il faut mettre en place certains utilitaires.

1.5 Contributions

Les contributions que nous avons réalisées dans ce travail sont en rapport avec plusieurs domaines liés aux systèmes distribués, à la sécurité des systèmes informatiques, et à l'utilisation des cartes à puce. Ce sont :

1. **La définition d'une architecture générale pour les grilles de cartes à puce** : nous avons proposé une architecture générique définie en termes de couches permettant de décrire les notions fondamentales pour le déploiement et la mise en place d'une infrastructure logicielle pour les grilles de cartes à puce. Cette architecture que nous avons voulue la plus proche possible des architectures classiques présente toutefois quelques aspects spécifiques liés aux propriétés des cartes, notamment à leur passivité, pour laquelle nous avons proposé des solutions concrètes.
2. **L'introduction de la notion de *Card Service*** : afin de permettre une meilleure intégration de la carte à puce dans les systèmes distribués, nous avons introduit la notion de Card Service, ce qui permet d'apporter une certaine facilité pour la communication avec la carte et l'exécution d'applications. Pour cela, nous avons proposé une architecture basée Card Service en se référant au modèle des Services Web [20].
3. **La mise en œuvre d'un environnement d'exécution sécurisé** : l'environnement d'exécution mis en place permet d'assurer la sécurité du support d'exécution, de l'application et des échanges. Les deux premiers aspects sont en fait garantis grâce aux procédures et aux mécanismes de protection intégrés aux cartes lors de leur fabrication. Pour ce qui concerne la sécurité des échanges, nous avons défini un protocole basé sur l'utilisation de certificats permettant à la carte et à son utilisateur de s'authentifier mutuellement et d'initialiser un canal sécurisé. Une des originalités de ce point est que l'utilisateur et la carte sont potentiellement distants et pas

[1] Cette architecture, développée au LaBRI antérieurement à cette thèse, consiste en un assemblage d'un nombre important de cartes, à la manière d'un cluster.

sur un poste de travail unique comme dans les configurations classiques d'utilisation des cartes à puce.

4. **La définition de quelques outils supplémentaires** : afin de mieux exploiter les ressources matérielles nous avons proposé trois outils pour l'utilisation des grilles de cartes à puce. Le premier est un outil de déploiement des applications cartes, le second est un outil pour la composition des *Card Services* et le troisième est un ordonnanceur d'exécution pour des applications parallèles.

1.6 Organisation du document

La suite de ce manuscrit est organisée comme suit :

- Dans le chapitre 2, nous présentons le concept des cartes à puce, les différents types de cartes qui existent, les utilisations de la carte, le protocole de communication avec la carte, etc. Nous introduisons également les cartes ouvertes et nous décrivons la technologie Java card, qui est une implémentation de ce type de cartes.
- Le chapitre 3 est consacré à la présentation du standard GlobalPlatform [67], est présent sur la plupart des cartes Java et qui fournit une solution pour les problèmes de sécurité liés aux cartes ouvertes, et en particulier pour la protection des échanges avec la carte. Ce chapitre sera aussi l'occasion de rappeler des notions de cryptographie que nous utiliserons par la suite.
- Dans le chapitre 4, nous présentons les infrastructures de type grille et les problèmes de sécurité posés par ces architectures, est qui sont dus à la mobilité du code qui doit être téléchargé sur le support matériel pour être exécuté. Nous définissons également une architecture logicielle générique pour les grilles de cartes à puce ainsi qu'une architecture basée sur la notion de *Card Service*.
- Dans le chapitre 5, nous présentons un prototype d'implémentation de la couche intergiciel permettant de gérer une grille de cartes à puce de type Java. Nous présentons les différents constituants de l'architecture et nous proposons des solutions au problème du déploiement des applications (accès aux cartes, passivité des cartes, etc.).
- Dans le chapitre 6, nous proposons une solution basée sur l'utilisation de certificats permettant d'assurer la sécurité des communications depuis et vers une carte à puce. Cette solution a été validée formellement à l'aide de l'outil AVISPA [7].
- Pour une meilleure exploitation de la grille, des outils supplémentaires ont été développés. Ces outils seront décrits dans le chapitre 7. Nous proposons en particulier un outil d'installation-désintallation d'applications Java Card, un outil de composition de Card Services et un ordonnanceur tolérant aux fautes pour l'exécution parallèle.
- Finalement, plusieurs applications de validation sont présentées dans le chapitre 8. Il s'agit d'une application pour la calcul de la fractale de Mandelbrot, d'une application de type *data-mining* sécurisé et d'une application basée sur du calcul itératif asynchrone.

Chapitre 2

Le concept de la carte à puce ; introduction aux Java Cards

Sommaire

2.1 Introduction

Depuis son apparition, la carte à puce ne cesse de conquérir de nouveaux marchés tout en gagnant la confiance du consommateur. Elle est considérée aujourd'hui comme un élément essentiel et incontournable de la vie de tous les jours. Ceci est dû non seulement au fait que l'on puisse la transporter, mais aussi au niveau de sécurité élevé qu'elle assure. Du point de vue évolution technologique, la carte à puce est passée par plusieurs étapes : de la carte à mémoire à la carte à microprocesseur, de la carte mono-application à la carte multi-application ce qui lui a permis de proposer de nouvelles fonctionnalitées.

Dans ce chapitre nous faisons le point sur le concept de carte à puce. Nous introduisons également les cartes multi-applicatives et nous présentons le standard Java Card qui représente une des plateformes majeures de cartes ouvertes.

2.2 La technologie carte à puce

La carte à puce se présente comme l'alliance de deux composants : un circuit électronique figé sur un support plastique. Le corps (plastique) de la carte a une taille normalisée ($85,6$ mm de longueur et $53,89$ mm de largeur). Grâce à son circuit intégré, elle est capable de stocker et de traiter des données. Elle propose les mêmes fonctions qu'un ordinateur classique, mais à une échelle réduite : les performances de la carte sont très limitées en comparaison de celles d'une machine classique. Néanmoins, la carte à puce permet d'assurer la sécurité des données et du code qu'elle contient ce qui constitue son principal avantage.

2.2.1 Les utilisations de la carte à puce

En plus de sa sécurité, la carte présente un autre avantage, sa petite taille, qui la rend portable et mobile. Les données et les informations qu'elle contient sont disponibles à n'importe quel endroit où se trouve le porteur de la carte. Vus les avantages qu'elle présente et sa facilité d'utilisation, elle est devenue aujourd'hui incontournable dans notre vie quotidienne. Ses principales utilisations [83] sont le domaine des télécommunications, le secteur de la santé (notamment la carte Vital en France [78], la future carte NetC@rds en Europe [52]), le domaine bancaire et monétaire, le secteur des transports, etc. La carte téléphonique prépayée était parmi les premières utilisations des cartes à puce. Il s'agit d'une carte à mémoire (voir section 2.3) qui contient un compteur. Ce compteur représente les unités disponibles sur la carte et est décrémenté lors de chaque utilisation dans une cabine téléphonique.

Le deuxième domaine d'application de la carte à puce est le domaine bancaire. L'utilisation des cartes de payement de type EMV [47] et les portes monnaies électroniques permettent de faciliter et de simplifier les transactions commerciales et les opérations monétaires. De ce fait, la carte bancaire est aujourd'hui très répandue.

Une autre application de la carte à puce est l'identification. Un secret peut être donné à la carte permettant à son possesseur de s'authentifier auprès de certains services. Ceci est par exemple le cas des cartes d'identification professionnelle ou d'accès à des locaux sensibles (tels que les centrales nucléaires), les cartes de télévision à péage, etc.

Néanmoins, le secteur où l'utilisation de la carte à puce est la plus significative reste celui des télécommunications avec l'utilisation massive des cartes SIM [49]. Aujourd'hui, et à travers les réseaux GSM, la carte à puce, sous format SIM, est utilisée à la fois comme un élément d'identification et un support applicatif. Un opérateur téléphonique pourra ainsi proposer à ses clients des services personnalisées (via la carte SIM).

Dans le cadre de nos travaux de recherche, nous présentons une application originale de la carte à puce où elle est considérée comme un nœud de calcul sécurisé.

2.2.2 Les principales normes pour les cartes à puce

Comme nous venons de le voir, la carte à puce est aujourd'hui présente dans plusieurs secteurs : transport, communication, santé, etc. Afin de normaliser les applications d'un domaine particulier, plusieurs standards ont vu le jour. On trouve par exemple la norme EMV [46] pour le domaine bancaire et la norme ETSI [50] pour les cartes SIM. La norme ISO-7816 [84] reste la norme de base qui permet de définir les caractéristiques physiques, matérielles et logicielles des cartes à puce. Elle spécifie les propriétés physiques de la carte (dimensions de la carte et de la puce), mécaniques, électriques et magnétiques. La norme ISO-7816 décrit aussi les différents contacts de la puce (voltage, entrées/sorties, etc.) ainsi que leur position. Les protocoles de transmission et de communication (respectivement au niveau transport et application) sont également définis par la norme ISO-7816.

2.2.3 Pourquoi a-t-on confiance en la sécurité de la carte à puce ?

Tout système ou produit issu des Technologies de l'Information peut être le sujet d'une procédure de certification, menée par un organisme habilité, à l'issue de la quelle un certificat (selon une norme bien définie) est délivrée. On dit alors que le produit est conforme à la norme considérée. Ce processus

de certification peut être appliqué pour une carte à puce. L'émetteur de la carte ou son utilisateur final (voir Annexe A) peuvent alors s'appuyer sur un tel certificat afin de renforcer le degré de confiance qu'ils auront dans la carte.

Le processus de certification consiste en une procédure d'évaluation sécuritaire menée par des organismes reconnus et réalisée dans un cadre officiel. Il s'agit d'une procédure relativement complexe dont l'objectif est d'établir le niveau de confiance que pourra avoir un utilisateur final en la sécurité d'un produit. Le produit doit donc être conforme à une norme donnée spécifiée et satisfait l'ensemble des règles de sécurité définies par cette norme.

En France, les cartes à puce sont évaluées par des CESTI (Centre d'évaluation de la Sécurité des Technologies de l'Information) et sont certifiées par la DCCSI (Direction Centrale de la Sécurité des Systèmes d'Information). Ainsi, un utilisateur final peut être certain qu'il possède une carte sûre, et qu'il ne court pas de risque majeur même si sa carte a été la cible d'attaques.

Il existe en effet plusieurs normes dans le contexte des Technologies d'Information. Citons par exemple la norme ISO/CEI 17799 [86] qui établit les principes généraux pour la mise en œuvre et l'entretien de la sécurité au sein d'un organisme. On trouve également la norme FIPS 140 [54] qui décrit un ensemble de règles de sécurité (telles que la gestion des clés, l'utilisation de certains algorithmes, etc.) que doit satisfaire un module cryptographique intégré à un système de sécurité. Il existe aussi la norme ISO-15408 [85] connue sous le nom de Critères Communs. Cette certification (ISO-15408) propose 7 niveaux d'assurance (EAL : *Evaluation Assurance Level*) fondés sur des tests fonctionnels et structurels, des vérification conceptuelles et des méthodes de validation formelle.

Plusieurs modèles de cartes (tels que Gemplus ou JCOP) disposent aujourd'hui de certificats qui concernent le processeur et le système d'exploitation de la carte [26].

2.3 Les différents types de cartes à puce

Il existe deux principales catégories de cartes à puce. Elles se différencient par leur architecture interne et leur mode de fonctionnement.

Du point de vue architecture interne, les premières cartes qui ont été conçues avaient une architecture fondée sur une unité de mémoire. La mémoire contenait un compteur qui se décrémentait après chaque utilisation de la carte (cas des cartes téléphoniques prépayées par exemple) : on parle de cartes à mémoire. Avec l'evolution de la technologie, on a pu embarquer un microprocesseur dans la puce, d'où le deuxième type : les cartes à microprocesseur.

- **La carte à mémoire** :
  Une carte à mémoire comporte simplement une puce mémoire. C'est la première génération de cartes à puce. Elle ne possède pas de microprocesseur, mais en revanche elle peut disposer d'un circuit électronique non reprogrammable. Les premiers modèles de cartes avaient une mémoire de l'ordre de quelques Ko (de 1 Ko à 4 Ko). Aujourd'hui cette taille peut atteindre les 4 Mo [101]. Ce type de carte est très peu utilisé et est en voie de disparition au profit des cartes à microprocesseur.
- **La carte à microprocesseur** : ce modèle de cartes est celui que les anglophones appellent *SmartCard*. Les composants fondamentaux d'un ordinateur sont reproduits sur la puce de la carte (voir figure 2.1) sur une surface ne dépassant pas les 25 mm^2 (norme ISO-7816). On y trouve un processeur, dont les capacités sont extrêmement limitées : bus de 8 bits et fréquence de 4,77 MHz pour les modèles standards. On y trouve également un co-processeur cryptographique auquel est associé un générateur de nombres aléatoires RNG (Random Number Generator). Ces différents processeurs communiquent par un bus avec trois types de mémoire :
  - la mémoire ROM (*Read Only Memory*), mémoire dont le contenu non modifiable inclut le système d'exploitation de la carte et éventuellement certaines applications et/ou des données persistantes ;
  - la mémoire vive RAM (*Random Access Memory*), mémoire de travail volatile (son contenu disparaît en l'absence d'alimentation électrique) et à accès rapide ;

– la mémoire persistante, mémoire de stockage, à contenu non-volatile mais à accès lent ; telles que par exemple l'EEPROM (*Electrically Erasable Read Only Memory*) ou les mémoires Fe-RAM (*Ferroelectric Random Access Memory*).

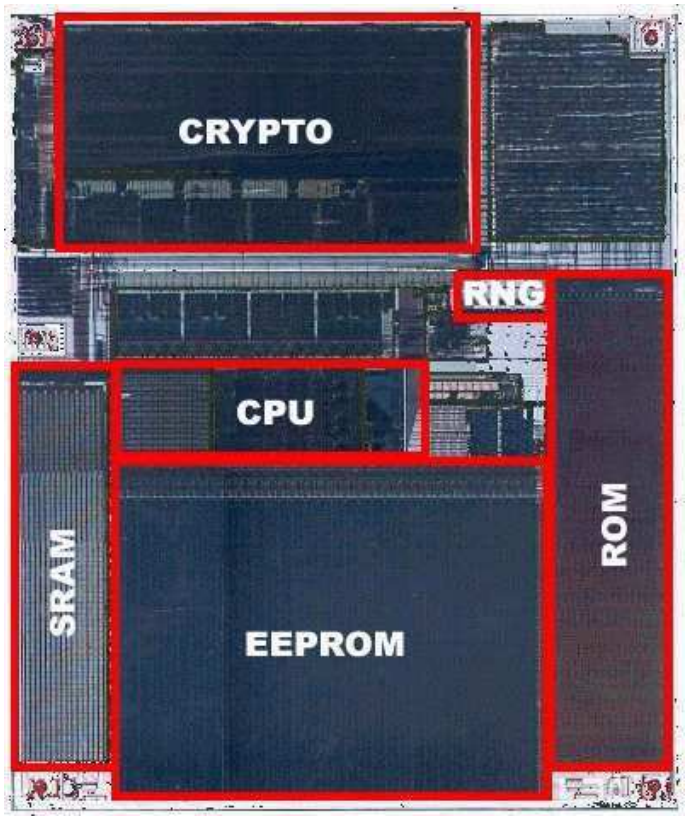


FIG. 2.1 – Architecture interne de la puce d'une carte à microprocesseur (source : inconnue)

La deuxième classification se base sur le mode de fonctionnement de la carte. De par sa conception la carte à puce ne dispose pas d'alimentation électrique propre et dépend donc d'un matériel appelé *Card Accepting Device* (CAD). Le CAD, appelé aussi lecteur de cartes, constitue l'interface d'entrée/sortie avec la carte. On distingue deux classes de cartes selon la nature du contact avec le CAD :

– **Les cartes à contact** : la carte doit être insérée dans un lecteur de cartes pour qu'elle devienne opérationnelle. Les cartes à contact offrent une communication série via huit contacts formant le micro-contact placé en dessus de la puce comme le montre la figure 2.2. C'est aussi à travers ce micro-contact que la carte se procure l'énergie nécessaire à son fonctionnement.
– **Les cartes sans contact** : avec ces cartes, plus besoin d'insérer sa carte dans un lecteur. Elles contiennent un émetteur hyperfréquence et communiquent via une antenne radio intégrée, qui assure aussi son alimentation, comme le présente la figure 2.3.

Les cartes qui autorisent ces deux modes de fonctionnement sont tout simplement appelées des cartes combi ou dual-interface.

2.4 Architecture d'une carte à puce

Les caractéristiques matérielles des cartes à puce sont très limitées. Ceci concerne aussi bien la taille de mémoire, que la puissance du processeur. Pour ce qui est de la mémoire, les cartes haut de gamme actuelles sont équipées de 512 Ko de ROM, 1 Mo de mémoire persistante et 16 Ko de RAM. Ces limitations sont dues aux contraintes de la taille des puces imposées par la norme ISO-7816. Dans

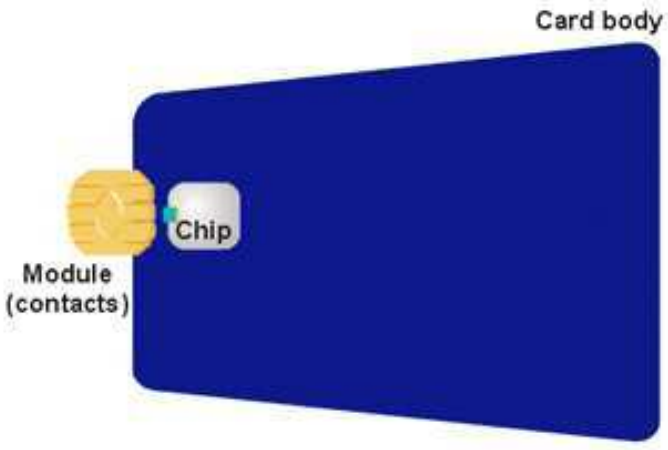


Fig. 2.2 – La carte à contact (source : [60])

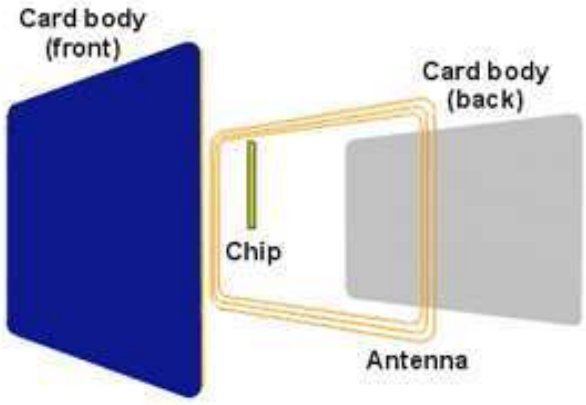


Fig. 2.3 – La carte sans contact (source : [60])

le tableau 2.1, nous présentons l'évolution des capacités des processeurs et des mémoires des cartes entre les années 1980 et 2000.

Année	Taille de bus	Fréquence	RAM	Mémoire persistante
1981	8 bits	4,77 MHz	36 octets	1 Ko
1985	8 bits	4,77 MHz	128 octets	2 Ko
1990	8 à 16 bits	4,77 MHz	256 octets	8 Ko
1996	8 à 32 bits	4,77 à 28,16 MHz	512 octets	32 Ko
2000	8 à 32 bits	4,77 à 28,16 MHz	1.5 Ko	64 Ko
2003-2004	8 à 32 bits	4,77 à 100 MHz	16 Ko	256 Ko

Tab. 2.1 – Évolution des caractéristiques des cartes à puces (d'après [24, 142])

L'utilisation de nouvelles technologies mémoires [142] (telles que la mémoire Flash, la FRAM, etc.) caractérisées par un taux d'intégration élevé, et qui sont d'ores et déjà utilisés pour les cartes à puce, permet d'augmenter considérablement la taille des mémoires. De plus, la gravure des circuits intégrés de plus en plus fine [142], laisse espérer une croissance importante de la des capacité des cartes.

Dans les deux dernières années, de nouvelles cartes à puce "géantes" [29] avec des capacités mémoires de l'ordre de 128 Mo ont été proposées sur le marché des cartes SIM. De plus, l'intégration de

nouvelles technologies a permis d'atteindre des capacités mémoire de 1 Go[1] et de disposer de deux processeurs sur la même carte. Malgré cette évolution, les capacités des cartes restent toujours limitées par comparaison avec celles des ordinateurs de bureau comme le montre le tableau 2.2.

	Carte à puce	Ordinateur de bureau	Ratio
RAM	1 Ko	128 Mo	130 000
Stockage	64 Ko	6 Go	100 000
Vitesse de transfert	192 Kbits	100 Mbits	500
Vitesse du processeur	20 Mips	500 Mips	25

TAB. 2.2 – Comparaison entre les performances d'une carte et celles d'un PC (source : [91])

2.5 Le protocole APDU

La carte à puce dépend d'un terminal pour être opérationnelle. Ce terminal constitue une interface d'entrée/sortie avec elle et permet de lui assurer un canal de communication avec les applications extérieures. Cette communication se fait sous forme d'unités APDU (*Application Protocol Data Unit*) dont la structure est définie par la partie 4 de la norme ISO-7816 [84]. Il existe deux catégories d'APDU : les commandes APDU et les réponses APDU. Les premières sont envoyées du terminal vers la carte et contiennent une commande à exécuter. Les secondes sont envoyées (comme réponse) de la carte vers le terminal.

La structure d'une commande APDU est présentée dans le tableau 2.3. Une telle commande est composée de :

- un en-tête de quatre octets :
  - *CLA* identifie la catégorie de la commande et de la réponse APDU ;
  - *INS* spécifie l'instruction portée par la commande ;
  - *P1* et *P2* sont deux octets utilisés pour paramétrer l'instruction ;
- un corps optionnel de taille variable :
  - *Lc* spécifie la taille du champ de données ;
  - le champ de données contient les données envoyées à la carte pour exécuter l'instruction spécifiée dans l'entête ;
  - *Le* spécifie le nombre d'octets (éventuels) attendus par le terminal et qui seront contenus dans le champ de données de la réponse APDU qui sera retournée par la carte.

En-tête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Champ de données	Le

TAB. 2.3 – Structure d'une commande APDU

La structure d'une réponse APDU est présentée dans le tableau 2.4. Elle est constituée :

- d'un corps optionnel de taille variable qui correspond au champ de données. La longueur des données peut être égale à la valeur du champ *Le* reçu dans la commande APDU ;
- une en-queue obligatoire de deux octets qui correspondant aux octets d'états ou *Status Word* (*SW1* et *SW2*). Ils précisent l'état de la carte après exécution de la commande APDU correspondante. Par exemple, "0x9000" signifie que l'exécution s'est déroulée avec succès.

Corps optionnel	En-queue obligatoire	
Champ de données	SW1	SW2

TAB. 2.4 – Structure d'une réponse APDU

[1] Il s'agit d'une mémoire étendue accessible en USB. Nous avons proposé par ailleurs dans [31] un mécanisme basé sur la swapping sécurisé de données permettant à la carte d'avoir une capacité mémoire plus étendue.

Les cartes à puce fonctionnent selon un modèle client/serveur dans lequel elles jouent le rôle du serveur. L'application qui se trouve sur l'entité extérieure est considérée comme le client. Ce mode de fonctionnement signifie que le code embarqué dans la carte est passif : c'est le code installé dans la station qui prend l'initiative d'appeler les méthodes dans la carte qui ne sait que répondre, et non l'inverse[2]. Ainsi, les deux types de structures de communication, commande et réponse, sont toujours couplées.

2.6 La sécurité des cartes à puce

La carte à puce peut être la cible d'attaques de diverses natures. Ces attaques sont classées selon deux grandes catégories : les attaques physiques et les attaques logicielles.

2.6.1 Les attaques physiques

Toute attaque menée sur le support matériel de la carte est appelée une attaque physique. Ce type d'attaque repose sur l'observation et/ou la modification des composants électroniques constituant la carte. Les attaques physiques peuvent être répertoriées en deux sous-classes.

Les attaques invasives : les attaques invasives consistent à agir directement sur les composants élémentaire de la puce (microprocesseur, mémoires et bus internes) par lecture, écriture, effacement ou altération des données. Les attaques invasives sont basées sur des techniques de déballage visant à isoler la puce des autres composants de la carte (le micro-contact et le corps plastique) ce qui permet d'accéder directement au *micro-chip*, comme le montre la figure 2.4.

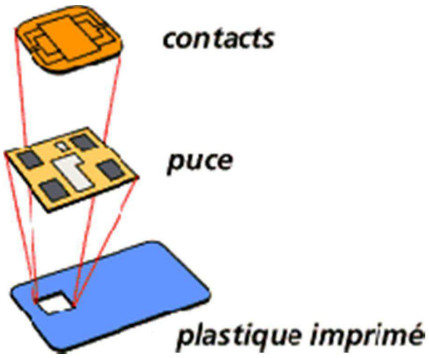


Fig. 2.4 – Isolation de la puce des autres composants de la carte (source : inconnue)

Après une telle attaque, la carte n'est en général plus utilisable. Parmi les attaques invasives, on trouve par exemple le *microprobing* [2, 98] qui consiste à exposer la puce à des ondes électromagnétiques afin d'observer l'information qui circule. D'autres formes d'attaques telles que les attaques par rétro-conception matérielle [3, 2] peuvent également être appliquées à la puce.

Vue la taille minuscule du *micro-chip* et de ses composants électroniques, les attaques invasives sont extrêmement délicates à réaliser : la moindre petite erreur peut conduire à endommager le circuit et donc à faire échouer l'attaque. De telles attaques requièrent par ailleurs l'utilisation d'un matériel sophistiqué tel que des microscopes électroniques ou des sondes FIB (*Focused Ion-Beam*) [2]. De plus, l'attaquant doit avoir des compétences dans la conception et la fabrication des circuits intégrés et de bonnes connaissances en micro-électronique. Il doit aussi disposer d'informations détaillées sur la cible avant de procéder à une telle attaque pour contourner les mécanismes de protection matérielle mis en place. Il est donc bien clair que les attaques invasives sont très dures à réaliser et ne sont pas à la portée de tout le monde.

[2]Dans une partie de nos contributions, nous avons conçu un mécanisme qui permet de contourner cette passivité (appelé mode pro-actif).

Les attaques non invasives : la seconde forme d'attaques physiques est constituée par les attaques non invasives. Ces attaques peuvent être qualifiées de discrètes par comparaison aux attaques invasives : les attaques non invasives ne mettent pas en péril la carte à puce. Elles sont menées à travers l'environnement extérieur de la carte pendant son activité sans pour autant l'endommager.

Les attaques non invasives peuvent être classées en deux catégories : les attaques basées sur l'observation de l'environnement de la carte, et celles basées sur sa perturbation [15].

- L'observation de l'environnement de la carte : les attaques par observation consistent essentiellement à déceler et à exploiter des canaux cachés pouvant éventuellement fournir de l'information utile sur l'état interne de la puce. Les grandeurs physiques émises par la carte pendant son activité sont détectées et mesurées. Ces mesures sont analysées et exploitées pour en déduire le comportement du microprocesseur et acquérir des informations sur les secrets de la carte.

  Actuellement, la mesure de temps d'exécution (*Timing Attack* [96]), ou l'analyse de consommation électrique (*Simple/Differential Power Analysis* [97]) représentent les attaques par canaux cachés les plus exploitées. L'analyse du rayonnement électromagnétique[3] engendré par le fonctionnement du microprocesseur (*Simple/Differential ElectroMagnetic Analysis* [119]) peut aussi être appliquée à la carte à puce.
- Les attaques par perturbation : ce type d'attaques consiste à perturber le fonctionnement de la carte, et notamment celui du microprocesseur. L'objectif est de mettre la carte dans un état anormal en agissant sur ses conditions d'utilisation habituelles. Ainsi, il sera peut être possible de modifier le contenu d'une zone particulière de la RAM et/ou des registres, ou de provoquer le décodage erroné d'une instruction. L'altération des paramètres opérationnels de la carte offre de nombreuses possibilités d'attaques. En appliquant des valeurs de tension ou de température hors de la plage autorisée, il est possible de perturber le fonctionnement du microprocesseur.

  Les attaques par perturbation s'inscrivent dans une classe d'attaques de type *DFA : Differential Fault Analysis* (attaques par injection de fautes). Ces attaques consistent à induire des fautes pendant l'activité de la carte provoquant une perturbation très brève du processeur [9, 12, 62]. Ce type d'attaque peut s'avérer très efficace si elle est réalisé lors de la manipulation de secrets (par exemple, lors d'une signature RSA [15]). Elles peuvent être réalisées par exemple par envoie d'impulsions lumineuses de très courte durée, ou en exposant la puce à différents types de rayonnement (tels que les rayons X ou les rayons UV) [12].

2.6.2 Les attaques logicielles

Outre les attaques matérielles, un agresseur peut aussi cibler le logiciel embarqué sur la carte (le système d'exploitation de la carte que les applications qui y sont installées).

Nous nous intéressons principalement aux cartes multi-applicative. Ces cartes sont caractérisées par la cohabitation de plusieurs applications et autorisent l'installation/désinstallation d'applications tout au long de leur cycle de vie. La possibilité de charger dynamiquement des programmes constitue une vraie menace pour la sécurité du système carte : il est en effet possible d'installer une application, de type cheval de Troie, contenant un code malicieux pouvant interagir avec le logiciel embarqué.

Un attaquant peut chercher à installer une application qui lit et vide le contenu de la mémoire (*memory dump*) [102], ou qui vise à modifier les données et les instructions des autres applications [61]. Un attaquant peut aussi essayer d'exploiter les failles de sécurité liées au partage d'objet et charger un code qui tente d'accéder frauduleusement aux fonctionnalités des autres applications à travers les services partagés [107]. Il est également possible qu'un agresseur essaie d'exploiter les failles de conception au niveau du système (machine virtuelle, mécanisme de séparation entre les applications) et des programmes. Avec plus ou moins de difficulté, l'attaquant peut espérer contourner le mécanisme de sécurité mis en œuvre et détourner l'application et/ou le système de leurs fonctionnalités initiales. Ces attaques sont généralement dues à une mauvaise spécification ou à un bug d'implémentation de

[3] Il s'agit des traces électromagnétiques qui résultent des flux de courant parcourant un circuit.

l'application ou du système d'exploitation. La sécurité de la carte à puce repose principalement sur l'efficacité et la sûreté de son système d'exploitation. Ceci concerne particulièrement le module de gestion de mémoire qui doit être robuste vis à vis des défauts d'alimentation qui peuvent se produire suite par exemple à une attaque par arrachage.

Notons enfin que les attaques matérielles et logicielles peuvent être menées conjointement, augmentant ainsi les chances de succès.

2.6.3 Contre-mesures

Pour s'opposer à ces attaques, des contre-mesures ont été mises en œuvre. Dans cette section nous présentons, pour chaque catégorie d'attaque, les principales contre-mesures associés.

Les attaques invasives

Dès les premières étapes de son cycle de vie, c'est-à-dire à partir de la conception et de la fabrication de la carte, des mécanismes de protection sont mis en œuvre permettant de contrecarrer les attaques invasives.

Avec l'évolution de la micro-électronique, les circuits intégrés bénéficient aujourd'hui d'une sécurité de haut niveau. Grâce aux technologies avancées de fabrication, les puces sont réalisées avec une intégration très poussée [15] : taille des composants élémentaires inférieure à un micron, plusieurs couches de métallisation, etc. D'autres techniques peuvent aussi être utilisées dont les plus courantes sont [79, 80] :

- détecteurs d'attaque invasive : la surface de la puce est recouverte par plusieurs types de capteurs permettant d'intercepter toute utilisation en dehors de la norme. Dans une telle situation, la carte peut se bloquer définitivement (ou même s'auto-détruire) afin de protéger les données et les informations qu'elle contient contre une éventuelle attaque invasive.
- dissimulation de conception : le schéma d'assemblage des différents composants élémentaires (mémoires, bus, etc.) de la puce n'obéit pas aux modèles classiques. Autrement dit les règles de conception des circuits intégrés ne sont plus respectées. Il est possible par exemple de placer les cellules mémoires dans un ordre quelconque, sans tenir compte de leurs adresses comme c'est le cas pour un disposition classique [15].
- enfouissement des bus : afin de rendre difficile l'accès aux bus internes, ces derniers sont enfouis dans différents niveaux de métallisation.
- chiffrement interne : le contenu des mémoires et les données qui circulent à travers les bus internes sont chiffrés. Le mécanisme de chiffrement/déchiffrement est géré par le microprocesseur de la carte.
- calcul d'intégrité : cette méthode consiste à vérifier périodiquement l'intégrité des données présentes sur la carte afin de s'assurer qu'elles n'ont pas été altérées par une entité extérieure.

Les attaques non invasives

Les attaques par canaux cachés sont basées sur l'observation et l'analyse de grandeurs physiques (tels que le temps d'exécution, la consommation en énergie, etc.). L'objectif est de déduire les secrets de la carte, notamment les clés de chiffrement/déchiffrement utilisées par la carte.

Aujourd'hui, ce type d'attaques est bien maîtrisé grâce à des implémentations spécifiques des algorithmes cryptographiques (par exemple les *S-Box* de AES [14, 99]). Une des contre-mesures les plus courantes est l'introduction d'un facteur aléatoire lors de la manipulation de données secrètes [81, 113]. Il est également possible d'appliquer des techniques de masquage de données (application de fonctions logiques de type XOR, NAND, etc. sur les données d'entrée) [1, 141]. D'autres techniques sont basées sur l'injection d'opérations supplémentaires ou la permutation des instructions ce qui permet de rendre les exécutions non déterministes [73].

Parallèlement à ces contre-mesures logicielles, des stratégies de protection peuvent aussi être appliquées au niveau matériel. Une première contre-mesure que l'on doit à l'avancée de la micro-électronique

et au perfectionnement des procédés de fabrication des *micro-chips*, est la réduction de la consommation en énergie des composants électroniques ce qui a pour effet de compliquer l'acquisition de mesures [111]. Les recherches en électronique sont orientées vers la conception de circuits intégrés caractérisés par une consommation en énergie quasiment constante indépendante des informations manipulées [48, 117]. D'autres contre-mesures consistent à éliminer l'harmonie et la régularité des émissions du circuit intégré, due à la fréquence d'horloge, en proposant des architectures asynchrones [87, 108].

Par ailleurs, en ce qui concerne les attaques par DFA, les contre-mesures portent principalement sur la vérification périodique de l'intégrité des données afin de s'assurer qu'aucune attaque n'est en cours.

Il faut noter enfin que les contre-mesures matérielles et logicielles peuvent s'appliquer conjointement.

Les attaques logicielles

Vu leur aspect ouvert, les cartes multi-applicatives nécessitent des mesures spécifiques pour contrecarrer les attaques qui visent le logiciel. Les attaques externes, liées au chargement de code malicieux, peuvent être contrecarrées par une étape de vérification de l'intégrité de l'application avant de la charger sur la carte. C'est ce que fait par exemple l'utilitaire *checker* fourni par Sun pour les cartes Java. Nous pensons que l'utilisation d'un standard tel que GlobalPlatform est indispensable pour assurer la sécurité des cartes ouvertes, le droit de charger et d'installer des applications étant alors restreint à des entités autorisées. Seuls l'émetteur de la carte et le fournisseur de l'applications, qui théoriquement doivent se faire confiance, peuvent installer et supprimer des programmes en respectant un protocole d'authentification.

Une fois une application installée, c'est le système d'exploitation de la carte qui doit assurer la sécurité de son exécution et la protection de son code. Pour ce qui concerne le cloisonnement des applications, il est possible de mettre en place un mécanisme d'solation qui assure une séparation complète entre les espaces mémoires des application de la carte, comme par exemple le mécanisme du pare-feu utilisé dans les cartes Java. Ainsi, si une application hostile réussit à franchir l'étape de vérification de son code, elle sera nécessairement bloquée par le mécanisme de séparation qui l'oblige à rester dans son espace mémoire et elle ne pourra donc pas perturber les autres applications installées, ni même endommager le système carte. Les attaques menées à travers les objets partagés peuvent être contrées si le partage est soigneusement défini.

Finalement, nous pensons que les fabricants et les concepteurs des cartes à puce détiennent d'autres contre-mesures, qui sont pour des raisons de concurrence, gardées confidentielles.

2.7 Vers des cartes ouvertes

Depuis la naissance des cartes à puce, le langage de programmation utilisé était l'assembleur. Les programmes contenaient à la fois le système d'exploitation et l'application. Ce processus de développement supposait la réalisation d'une série de tests coûteux et de très longue durée avant que le programme ne soit figé (phase de masquage) sur la ROM de la puce [24]. Une fois figé, toute évolution ou modification du code demeurait difficile voir impossible. Ce modèle de carte est appelé cartes monoapplicative. De plus, puisque le langage assembleur dépend fortement du processeur sous-jacent , il n'est pas possible de réutiliser un même programme pour des cartes provenant de plusieurs constructeurs et disposant d'une plate-forme matérielle différente. Par conséquent, un tel schéma est inadapté pour certains domaines, pour lesquels un temps de mise sur le marché (*time to market*) court est primordial. Ce n'est qu'au milieu des années 90 que les recherches ont abouti à une nouvelle génération de cartes permettant de simplifier énormément le processus de développement et donc de réduire considérablement le temps de déploiement des applications. Cette nouvelle génération de cartes est basée sur un système ouvert autorisant le chargement dynamique d'applications après leur mise en circulation et permettant la cohabitation de plusieurs programmes (pouvant éventuellement communiquer entre eux) : on parle de cartes ouvertes. Des réalisations telles que Java Card [90] (dont la version 3.0 vient

de sortir le 31/03/2008), Multos [109], Smartcard.NET [130] ou BasicCard [10] font partie de cette nouvelle génération.

Les cartes ouvertes utilisent un système d'exploitation afin d'améliorer la portabilité des programmes, la compacité du code et la sécurité des applications. Ces systèmes d'exploitation sont constitués d'un environnement d'exécution et d'une machine virtuelle qui permettent de mettre en place des caractéristiques sécuritaires (telles que la séparation entre les applications) et d'exécuter les programmes. Les applications chargées sur la carte sont interprétées par la machine virtuelle afin qu'elles puissent s'exécuter. Le système d'exploitation de la carte possède un unique *thread* : les applications ne peuvent pas s'exécuter en parallèle.

Toutefois, le vrai défi pour les cartes ouvertes reste d'assurer (avec une architecture beaucoup plus ouverte et complexe) le même niveau de sécurité et de qualité que pour les cartes mono-applicatives.

2.8 Introduction à la technologie Java Card

Java Card est une des plate-formes de cartes multi-applicatives ouvertes. Lancé en 1997, Java Card est aujourd'hui le système d'exploitation pour carte à puce le plus utilisé dans le monde [45]. En 2007, la Java Card représentait 55% du marché des cartes à puce pour les différents applications (domaine des communications, secteur bancaire, etc.). Ce nombre était de 33% en 2005. Les estimations prévoient d'atteindre une valeur de 57% en 2009 et 49% en 2011. Pour le marché de cartes SIM, Java Card représente 43% du marché en 2004, contre seulement 24% en 2003 [6].

Le standard Java Card est constitué d'un sous-ensemble du langage Java. L'architecture de la machine virtuelle a été aussi modifiée du fait des restrictions de ressources matérielles, notamment la capacité mémoire limitée des cartes à puce.

La machine virtuelle Java Card [136], en occurrence la JCVM (*Java Card Virtual Machine*), est séparée en deux parties : une première partie qui s'exécute en dehors de la carte dont le rôle est d'effectuer des transformations sur ce code (dans un but d'optimisation afin de pouvoir le charger sur des périphériques à capacités limitées telles que les cartes à puce), et une deuxième partie embarquée sur la carte et qui permet de charger le code et enfin de l'exécuter.

Une des caractéristique de la technologie Java Card est la définition d'un environnement d'exécution, le JCRE (*Java Card Runtime Environment*), chargé de fournir des mécanismes de sécurité qui permettent une séparation entre le système de la carte à puce et les applications et entre les applications elles-mêmes [135]. C'est le JCRE qui encapsule la complexité sous-jacente et les détails du système de la carte. Les applets demandent des ressources et des services système au JCRE à travers les API Java Card [134]. Le standard Java Card définit aussi deux nouveaux formats de fichiers binaires qui apportent l'indépendance vis à vis de la plate-forme de développement :

- Le fichier ***CAP*** (pour *Convert APplet*) : les applications Java Card portent le nom d'*applet* et sont représentées par des fichiers *.cap*. Les fichiers *.cap* ont une syntaxe beaucoup plus simple que les *class files* classiques et qui est adaptée pour les matériels à ressources limitées tels que les cartes à puce. D'un point de vue technique, le fichier *cap* est une représentation binaire (de type archive utilisant le format *Jar*) qui contient les classes, les méthodes et les imports d'un paquetage Java. À chaque composant correspond un fichier dont le nom a pour extension *.cap*. Le format de chacun de ces fichier suit le format "*Tag, Length, Value*" (*TLV*).
- Le fichier ***EXP*** pour *EXPort* : ce fichier est utilisé par la partie de la machine virtuelle Java Card hors carte. Les fichiers *EXPort* (*.exp*) ne sont pas chargés dans la carte et ne sont pas directement utilisés par l'interpréteur, la JCVM embarquée. Ils sont produits et utilisés dans un but de vérification et d'édition des liens. Un fichier *export* contient les informations publiques sur les API d'un paquetage donné. Ce fichier peut être librement distribué par le développeur d'applets sans soucis de révéler les détails internes de l'implémentation.

Depuis Avril 2008, Sun a sorti une nouvelle version Java Card, la 3.0. Les spécifications de ce standard sont séparées en deux éditions ; une édition classique et une édition connectée :

– L'édition classique compatible avec la Java Card 2 et supporte les applets traditionnelles. L'edition classique présente aussi quelques évolutions par rapport à la plate-forme java Card 2.2 et dont l'architecture matérielle est semblable à celle requise pour le standard Java Card 2.2.

– L'édition connectée fournit une nouvelle machine Virtuelle qui intègre des nouvelles aspects tout en restant compatibles avec les applications développées pour l'édition classique. Parmi les nouveautés introduites dans l'édition connectée, nous pouvons citer :
  • une nouvelle machine virtuelle supportant le *multi-threading*, la ramasse miette (*garbage collector*), le *class loading*,...
  • des aspects orientés réseau, notamment une pile TCP/IP nécessaire pour les protocoles client/serveur tel que HTTP,
  • l'édition connectée propose un modèle pour les applications Web (en plus des des applets classiques) à travers une API pour le développement de servlets,
  • des structures de données telles que les *enumeration*, les *iterator*, etc.

Nous nous intéressons dans la suite de ce chapitre aux spécifications Java Card 2. la La plate-forme Java Card est basée sur le langage Java Card, la machine virtuelle Java Card, l'environnement d'exécution et un ensemble de bibliothèques accessibles par des API. L'architecture globale d'une Java Card est présentée sur la figure 2.5.

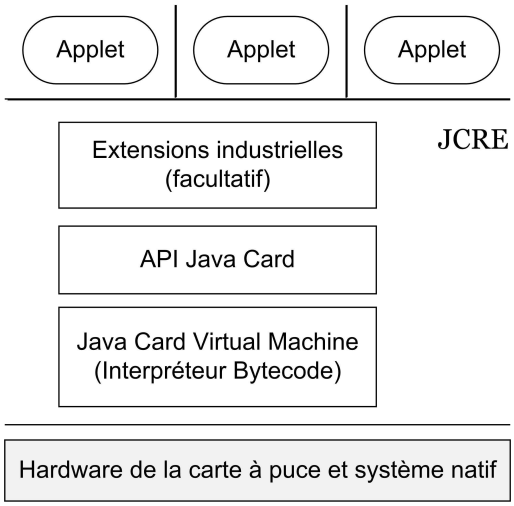


FIG. 2.5 – Architecture de la Java Card (d'après [35])

2.9 Le langage Java Card

À cause des contraintes liées à la restriction des capacités matérielles de la carte, certaines fonctionnalités présentes dans le langage Java, ne le sont pas dans Java Card. Toutefois, les caractéristiques fondamentales de Java ont été conservées. À l'instar de Java, Java Card est un langage objet sans pointeur fortement typé, qui autorise l'héritage de classe simple (*extends*) et l'héritage d'interface multiple (*implements*) et qui supporte les exceptions. De plus, les applications Java Card sont développées selon la syntaxe Java et sont compilées dans un premier temps par un compilateur Java classique. Les fichiers *classfile* (*.class*) résultats de la compilation sont ensuite injectés dans un convertisseur qui construit le fichier *.cap* directement téléchargeable sur la carte. Nous détaillons par la suite ces aspects dans la section 2.10.1.

À cause des limitations mémoire, les type de données larges (tels que les *float*, *double*, etc.) ne sont pas supportés par les Java Cards. Ceci est aussi le cas pour les chaînes de caractères (le type *String*)et les tableaux à plusieurs dimensions. Le *multi-threading*) et la sérialisation ne sont également

pas supportés par la technologie Java Card. La gestion du type entier étendu (*int*) et le ramasse miettes (*garbage collector*) sont des caractéristiques optionnelles

Nous présentons sur le tableau 2.5 les principales différences de Java Card par rapport au langage Java.

Caractéristiques Java supportées	Caractéristiques Java non supportées
▷ Type simple de donnée de petite taille : *byte*, *boolean*, *short* ▷ Tableaux à une dimension ▷ Notions de paquetages Java, classes, interfaces, exceptions, etc. ▷ Mots clés *extends*, *implements*, *super*, etc.	▷ Type simple de donnée de grosse taille : *long*, *double*, *float* ▷ Tableaux à plusieurs dimensions ▷ Chaînes de caractères ▷ *Threads* ▷ Sérialisation d'objet
Caractéristiques Java optionnelles	
▷ Le mot clé *int* (entiers sur 32 bits) ▷ Le ramasse miettes (*garbage collector*) (à partir de la version 2.2)	

Tab. 2.5 – Caractéristiques Java supportées et non supportées par Java Card (d'après [35])

2.10 La machine virtuelle Java Card

La machine virtuelle Java Card (la JCVM) fournit pratiquement les mêmes fonctionnalités que la machine virtuelle Java JVM. La différence principale est que la JCVM est découpée en deux parties comme le montre la figure 2.6 : une partie embarquée sur la carte et qui inclue l'interpréteur (et éventuellement l'installeur), et une deuxième partie hors carte, qui comprend le convertisseur.

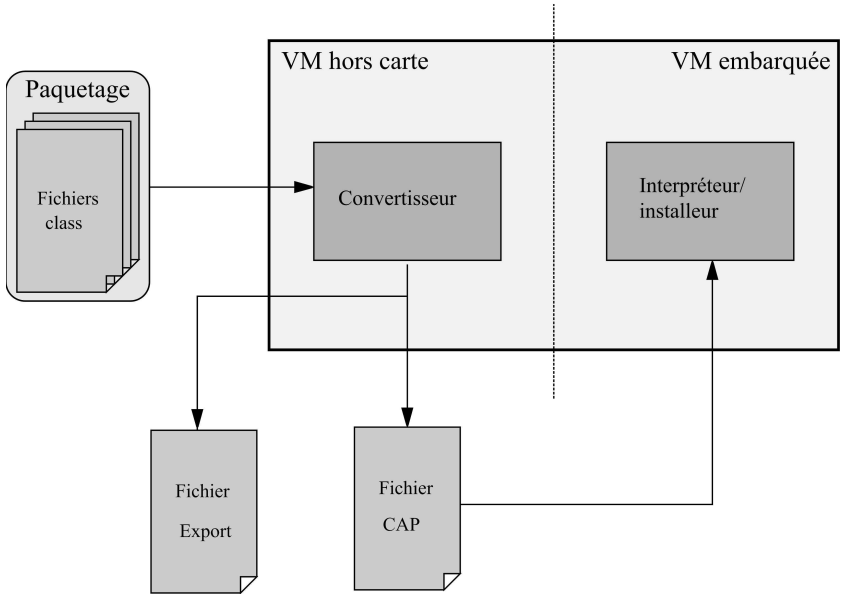


Fig. 2.6 – La machine virtuelle Java Card : JCVM (d'après [35])

Ce découpage en deux parties est dû principalement à la faible capacité mémoire de la carte, notamment en mémoire non volatile ce qui permet de gagner de la place en mémoire et de la consacrer uniquement à l'exécution des applications.

2.10.1 La JCVM hors carte

La JCVM hors carte a pour rôle de créer une représentation *.cap* de l'application qui soit directement téléchargeable sur la carte. Dans une première étape, le code source est compilé par un compilateur Java en utilisant les bibliothèques spécifiques à Java Card. Cette étape permet de repérer toute utilisation interdite de méthode, de classe et d'exception. Dans une seconde étape, les fichiers *.class* subiront une transformation dans le but d'optimiser la représentation du *bytecode*. Le point principal dans cette transformation, est l'utilisation d'un utilitaire checker, fourni par Sun, permettant de vérifier que le *bytecode* produit est bien un *btytecode* Java Card authentique, c'est-à-dire qu'il n'a pas été modifié pour faire introduire du code malicieux. Si l'application importe des classes provenant d'autres paquetages, il faut aussi charger les fichiers *export* associés à ces paquetages. Le procédure de conversion produit un fichier portant l'extension *.cap* et un fichier *export* représentant les API publiques du paquetage converti comme le montre la figure 2.7.

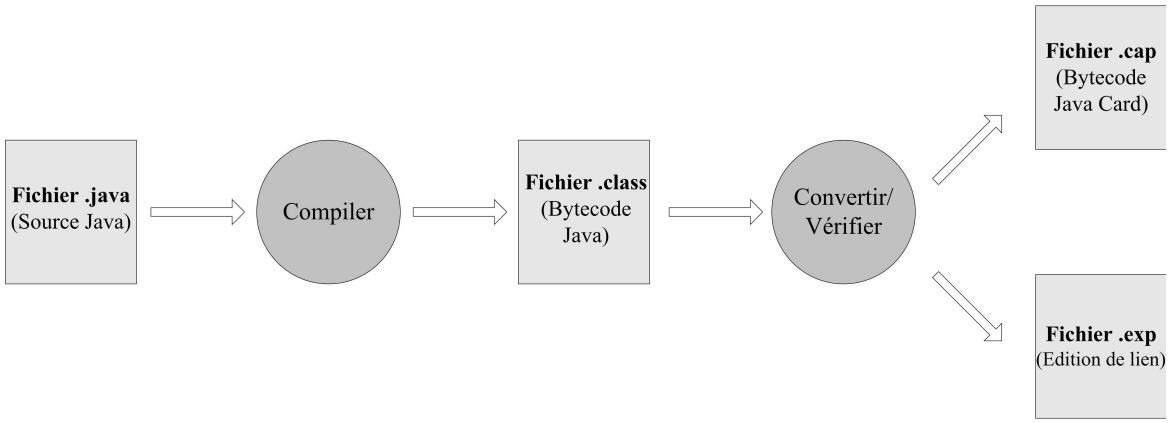


FIG. 2.7 – Schéma de développement d'applets Java Card

2.10.2 La JCVM embarquée

La JCVM embarquée peut être considérée comme la composition de deux unités.

L'interpréteur : L'interpréteur fournit un support d'exécution pour les applications indépendant du *hardware* de la carte. Il exécute des instructions *Bytecode* et contrôle l'allocation et la création des objets dans la mémoire.

L'installeur : La procédure d'installation consiste à décomposer un fichier *.cap* en une suite de commandes APDU envoyées à la carte et traitées par l'installeur. Ce dernier alloue l'espace mémoire nécessaire et crée une représentation mémoire du paquetage. Le fichier *.cap* est ainsi écrit dans la mémoire de la carte. Il faut préciser ici que l'installeur est un composant optionnel ; lorsque il n'est pas présent, la carte n'est simplement pas capable de recevoir de nouvelles applications après sa mise en circulation[4].

2.11 Les API Java Card

Les API Java Card sont un ensemble de classes optimisées pour la programmation des cartes à puce et qui respectent la norme ISO-7816 [84]. Du fait que la carte Java ne dispose pas de périphérique (écran, clavier, carte réseau, etc.), beaucoup de classes du standard Java (notamment celles relatives à l'affichage, au réseau, etc.) ne sont pas disponibles, car pas utiles, sur la plate-forme Java Card.

[4]La plupart des cartes Java sont compatibles GlobalPlatform [67]. La fonction d'installation/désinstallation des applications est donc assurée par l'entité "gestionnaire de carte" définie dans les spécifications GlobalPlatform.

Les API Java Card incluent essentiellement des classes pour gérer le noyau fonctionnel des applets Java Card et des services de cryptographie. Elles contiennent aussi des classes créées spécialement pour le standard ISO-7816. Depuis la version 2.2, d'autres classes sont ajoutées pour supporter le JCRMI. Parmi les principaux paquetages fournis par l'API Java Card, on trouve :

- le paquetage `javacard.lang` : contient les classes fondamentales pour le langage Java Card. Il fournit les classes `Object`, `Throwable` ainsi que différentes classes pour la gestion des exceptions (`NullPointerException`, `ArrayIndexOutOfBoundsException`, etc.) ;
- le paquetage `javacard.framework` : ce paquetage fournit un ensemble de classes et d'interfaces pour le développement et la gestion des applets Java Card et pour la communication avec la carte. On y trouve par exemple l'interface `ISO7816` définie pour le standard ISO 7816, la classe `Applet` qui doit être héritée par chaque applet Java Card. On trouve aussi la classe `APDU` qui permet à l'application carte de communiquer avec l'entité extérieure ;
- le paquetage `javacard.security` : comprend des classes et des interfaces pour l'implémentation de fonctions cryptographiques pour la plate-forme Java Card.

D'autres paquetages d'extension sont aussi définis par l'API Java Card. On trouve par exemple le paquetage `javacardx.biometry` qui fournit des fonctionnalités pour la biométrie, le paquetage (`javacardx.crypto`) pour des aspects cryptographiques (clés, fonctions cryptographiques, etc.), etc.

2.12 L'environnement d'exécution Java Card

L'environnement d'exécution de la Java Card, le JCRE, constitue l'ensemble des composants du système qui tourne à l'intérieur de la carte. Il est responsable de la gestion des ressources de la carte, de l'exécution des applets et de leur sécurité. De plus, le JCRE fournit les fonctionnalités d'exécution suivantes :

- La gestion de cycle de vie d'une applet Java Card : après le chargement d'une applet sur la carte à puce, l'environnement d'exécution peut agir sur celle-ci à travers un ensemble de méthodes. Pour cela chaque applet hérite de la classe `Applet`. Une fois installée, une applet peut être sélectionnée ou désélectionnée. Le JCRE est aussi responsable des communications avec le terminal. Ces communications se font sous forme d'APDU dont la structure est définie par la partie 4 de la norme ISO-7816 [84]. Le JCRE se charge des buffers d'entrée/sortie pour les APDU, et de leur acheminement vers l'applet appropriée.
- Le mécanisme du ***Firewall*** : la sécurité du code à l'intérieur d'une Java Card repose sur un modèle de pare-feu (*firewall*) qui définit une politique de séparation entre le système de la carte à puce et les applications embarquées et entre les applications elles-mêmes, comme illustré dans la figure 2.8. Lorsqu'un paquetage est chargé, un contexte lui est attribué. Par la suite, chaque objet est créé dans le contexte de son propriétaire. Deux applets dont les classes sont déclarées dans le même paquetage partagent un même contexte (dit contexte de groupe) et peuvent par conséquent communiquer. De ce fait, l'exécution d'une applet hostile ne peut pas affecter les applets des autres paquetages chargées sur la carte. En effet l'exécution d'une applet se fait toujours dans le contexte de groupe auquel elle appartient.
- Le partage d'objets : les différentes classes d'un même contexte peuvent échanger librement des informations, ceci n'étant pas possible pour deux applications appartenant à deux contextes différents. Pour cela Java Card introduit la notion d'objets partagés qui sont définis comme des classes héritant de l'interface `Shareable`. Lorsqu'une classe est définie comme partagée, elle autorise les applets de contextes différents à invoquer les méthodes définies dans son interface partageable sans qu'elles ne soient bloquées par le pare-feu, comme le montre la figure 2.8. Un tel mécanisme est réalisé sous le contrôle du JCRE qui veille à ce que les règles de partage soient respectées. Le mécanisme de partage d'objets permet donc d'établir une communication inter-applications en dépit du fait qu'elles ne partagent pas un même contexte
- Atomicité et transactions : avec la technologie Java Card, il est possible de rendre atomique un ensemble d'opérations en formant une transaction. Le début d'une transaction se fait avec un appel à la méthode `JCSystem.beginTransaction`. Il existe plusieurs façons de terminer normalement une transaction : soit en validant les effets de ses opérations (méthode

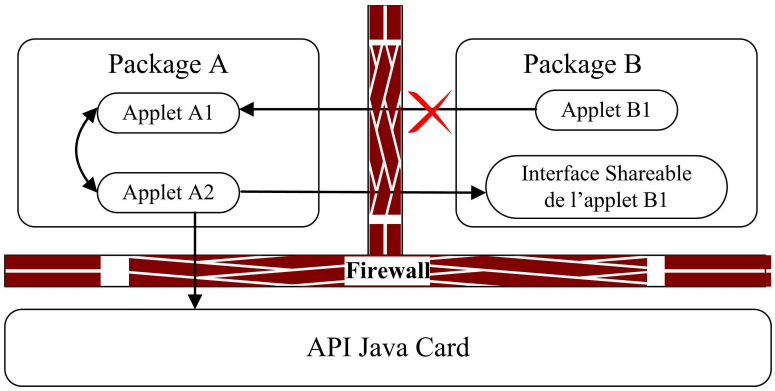


FIG. 2.8 – Éléments de sécurité pour le logiciel dans une carte Java

`JCSystem.commitTransaction`), soit en les annulant (méthode `JCSystem.abortTransaction`). Si pour une raison ou une autre, la transaction n'est pas finie, le JCRE, quant il reprend la main, la clôt avec un *abort*.

- Objets transients (ou temporaires) : afin d'améliorer leur performance, les applets peuvent créer des objets dont les données sont placées dans la mémoire volatile appelés des objets transients dont les champs ont un contenu temporaire. Pour cela, la technologie Java Card fournit à travers les API des méthodes permettant de créer de tels objets (notamment des tableaux). Lors de la création d'objets transients il faut préciser l'événement qui va déclencher la réinitialisation du contenu de l'objet (`CLEAR_ON_DESELECT` ou `CLEAR_ON_RESET`).
- Invocation de méthodes à distance : introduit depuis la version 2.2 de Java Card, le JCRMI (Java Card Remote Method Invocation) fournit un mécanisme pour qu'une application cliente, s'exécutant sur un entité extérieur, puisse appeler une méthode sur un objet distant présent sur la carte. En faisant abstraction du protocole APDU, JCRMI offre à l'utilisateur un moyen de communication transparent avec la carte, beaucoup plus simple que les échanges APDU. La gestion des commandes APDU de type RMI est assurée, côté carte, par un sous-ensemble embarqué qui permet d'appeler la méthode appropriée sur l'objet correspondant.

2.13 Conclusion

Au fil du temps, la carte à puce a subit des évolutions majeures et qui lui ont permis de gagner la confiance des utilisateurs. L'apparition des cartes ouvertes est une étape marquante de cette évolution. Avec cette nouvelle génération de cartes, il est désormais possible de télécharger dynamiquement des programmes sur une carte même après sa mise en circulation. De ce fait, des problèmes de sécurité liés à l'installation/désintallation des applications sont apparus. Parmi les solutions existantes, on trouve l'intégration d'extensions industrielles appropriées telles que le standard GlobalPlatform, spécialement conçu pour les cartes ouvertes et qui est souvent considéré comme incontournable pour ce type de cartes.

Chapitre 3

Le standard GlobalPlatform

Sommaire

3.1 Introduction

Avec une architecture beaucoup plus ouverte que leurs prédécesseurs, les cartes multi-applicatives offrent la possibilité de charger/supprimer des programmes tout au long de leur cycle de vie. Ceci engendre des problèmes liés à la gestion et aux droits d'installation/désinstallation des applications. Dans ce chapitre, nous présentons une solution, le standard GlobalPlatform, créé par un consortium portant le même nom, qui fournit un environnement dédié pour les cartes ouvertes permettant de gérer la coexistence de plusieurs applications provenant de plusieurs fournisseurs. Afin d'expliquer les procédures de sécurité mises en œuvre, nous rappelons quelques notions fondamentales de cryptographie.

3.2 Aperçu général des spécifications *GlobalPlatform*

Au milieu des années 1990, la technologie des cartes à puce a connu une évolution majeure avec l'apparition d'une nouvelle génération de cartes capables d'héberger et de gérer plusieurs applications simultanément. Ces nouvelles cartes, dites multi-applicatives, permettent également à l'utilisateur de charger et de supprimer des applications dynamiquement. Ceci a entraîné des problèmes de gestion du contenu de ces cartes ouvertes. Le consortium GlobalPlatform (anciennement OpenPlatform) qui regroupe des acteurs industriels et gouvernementaux, a proposé une solution pour ce type de cartes : *GlobalPlatform Specifications* [70]. Ces spécifications permettent de définir un standard qui est indépendant du matériel, des vendeurs, des fournisseurs et des applications et qui peut facilement être intégré sur tout type de carte ouverte.

Les acteurs de GlobalPlatform fournissent un ensemble complet qui consiste en une spécification pour les cartes [67], les terminaux [68] et les systèmes (tels que le processus de personnalisation, la gestion des clés, etc.) [69]. Seule la partie qui concerne les cartes nous intéresse. Nous décrivons dans ce chapitres quelques aspects de la *GlobalPlatform Card Specification* [67].

3.2.1 Présentation de *GlobalPlatform*

GlobalPlatform définit un environnement intégré pour le développement et l'exécution des applications sur les cartes à puce multi-applicatives. Ceci concerne principalement la gestion du processus de personnalisation de la carte et le chargement/désinstallation des applications. Un ensemble de règles est spécifié pour les communications avec le terminal, mais aussi pour les communications se déroulant à l'intérieur de carte.

3.2.2 Architecture d'une carte respectant *GlobalPlatform*

Une carte GlobalPlatform est constituée de composants interfacés par une API standard offrant une abstraction par rapport au matériel et au vendeur. Le système de gestion hors carte (telle que par exemple la procédure d'installation d'applets) est spécifié à travers plusieurs types de messages APDU.

Un exemple de configuration pour une carte GlobalPlatform est donné à la figure 3.1. Cette configuration comprend une application de l'émetteur de carte (*Card Issuer*), une ou plusieurs applications provenant des partenaires de l'émetteur de carte ou des fournisseurs d'applications (*Application Provider*), et une ou plusieurs applications proposant des services globaux tels que les services CVM : *Cardholder Verification Management*. Toutes les applications sont implémentées dans un environnement d'exécution sécurisé, le *Runtime Environnement* (RTE). Outre ces composants on distingue aussi des applications spéciales conçues pour la gestion de la sécurité (notamment pour la création d'un canal de communication sécurisé avec une entité extérieure) appelées domaines de sécurité (*Security Domains*).

L'environnement d'exécution (RTE) : GlobalPlatform est destiné à être intégré à n'importe quel environnement d'exécution sécurisé pour carte à puce multi-applicative. Le RTE assure une séparation complète entre les applications embarquées et assure également la gestion des communications avec l'extérieur.

L'environnement GlobalPlatform (OPEN) : les principales fonctionnalités de l'environnement GlobalPlatform (appelé OPEN pour Open Platform ENvironment) peuvent se résumer ainsi :

- la sélection des applications et l'acheminement des commandes APDU. Quand une commande *Select* est reçue, OPEN marque l'application référencée dans la commande de sélection comme l'application couramment sélectionnée, c'est-à-dire comme application active. Les commandes qui suivrait seront expédiées vers cette application.
- la gestion des canaux logiques. Le principe des canaux logiques est apparu dans les versions récentes de GlobalPlatform [65, 66]. En utilisant ce concept, plusieurs applications peuvent être sélectionnées de façon concurrente. Ceci peut également être utilisé pour une même applet afin qu'elle soit sélectionnée plusieurs fois sur différents canaux logiques grâce à un mécanisme

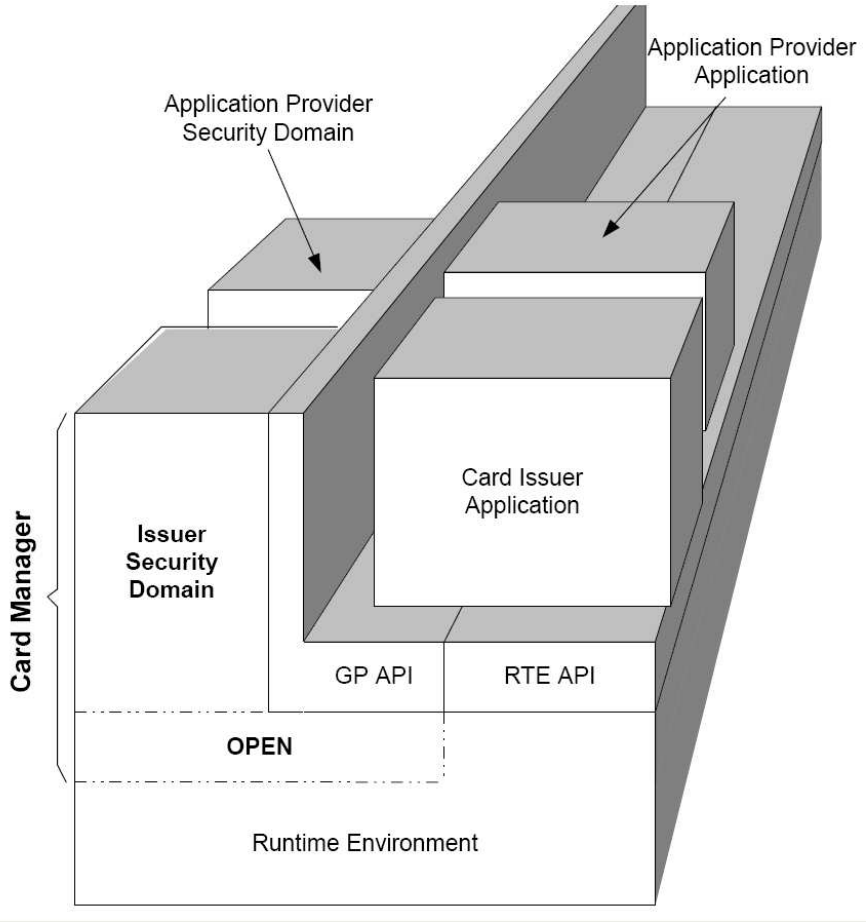


Fig. 3.1 – Architecture d'une carte GlobalPlatform (source : GlobalPlatform [65])

appelé multi-sélection. Il s'agit en fait d'une fonctionnalité facultative qui peut être omise lors de l'implémentation des spécifications GlobalPlatform.
– la gestion du contenu de la carte : la fonction de chargement et de désinstallation des applications est également assurée par l'environnement GlobalPlatform.

Pour pouvoir assurer ces fonctions et gérer le contenu de la carte, OPEN utilise un catalogue interne appelé *GlobalPlatform Registry*. Il contient les informations sur les codes exécutables chargés, les applications présentes, leurs associations avec les domaines de sécurité, etc.

Les API *GlobalPlatform*. Les API GlobalPlatform présentes sur la carte fournissent aux applications un ensemble de fonctions fondamentales, notamment pour la gestion d'un canal sécurisé. D'autres classes permettent de rendre accessible un ensemble d'informations gérées par l'environnement OPEN (état de la carte/application, données historiques, etc.)

Le *Card Manager* . Le *Card Manager* agit comme administrateur central de la carte, assure des fonctionnalités variées, et contrôle le système global de sécurité sur la carte. Il fournit des méthodes pour la vérification du détenteur de carte, accessibles à travers les API. De plus, et en tant que *Issuer Domain Security*, le *Card Manager* est considéré comme le représentant sur la carte, de l'émetteur de celle-ci. Cette fonction lui permet d'assurer la politique de sécurité adoptée par l'émetteur, notamment pour l'installation/désinstallation des applications. Pour cela, le *Card Manager* inclut aussi les fonctionnalités des domaines de sécurité que nous présentons juste après.

Les *Security Domains* : un domaine de sécurité peut être défini comme le représentant, sur la carte, du fournisseur d'applications. Il propose aux applications qui lui sont associées un ensemble de services cryptographiques tels que chiffrement/déchiffrement de messages, génération/vérification de signatures, etc. Chaque domaine de sécurité possède ses propres clés qui sont isolées de celles des autres domaines de sécurité, et en particulier de celles du *Issuer Domain Security*. En outre, un domaine de sécurité fournit une implémentation d'un *Secure Channel Protocol* afin de sécuriser les messages échangés entre les applications de la carte qui sont sous son contrôle et l'entité extérieure. Une application peut obtenir une référence sur son domaine de sécurité, et donc avoir accès aux services qu'il propose, à travers les API GlobalPlatform. Les domaines de sécurités peuvent être considérés comme des applications classiques et par conséquent ils en possèdent les mêmes caractéristiques (AID, cycle de vie, etc.). Leurs fonctionnalités sont en revanche spécifiques.

Depuis la version 2.2 [66], GlobalPlatform définit deux nouveaux composants :
- *Global Services application* : il s'agit d'une application fournissant des services globaux et pouvant être accessible par d'autres applications.
- *Trusted Framework* : doté d'un statut spécial, ce *framework* de confiance propose des services de communication interne pour les applications installées sur la carte.

3.2.3 La communication sécurisée

Les spécifications GlobalPlatform proposent des mécanismes et des procédures pour mettre en place une communication sécurisée durant laquelle les données échangées entre une carte et l'entité extérieure seront chiffrées. L'établissement d'un canal sécurisé est toujours précédé d'une étape d'authentification mutuelle. GlobalPlatform propose deux protocoles pour réaliser cette l'authentification. Le premier est basé sur des clés symétriques et le second sur des clés asymétriques.

Dans la suite de ce chapitre nous nous intéressons principalement à la partie de GlobalPlatform liée à la sécurité des communications, c'est-à-dire à l'établissement des canaux sécurisés. Nous rappelons auparavant quelques notions liées à la sécurité des informations, notions sur lesquelles s'appuie tout protocole de communication sécurisé.

3.3 Cryptographie et sécurité informatique

Définition de la cryptographie : Dans [106], la cryptographie est définie comme suit :
Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication.

Afin de mettre en œuvre les différents points de cette définition, des algorithmes cryptographiques doivent être utilisés.

3.3.1 Les procédés de cryptographie

Il existe deux catégories de procédés cryptographiques : les algorithmes à clé symétrique et ceux à clé asymétrique.

3.3.1.1 La cryptographie symétrique

Les algorithmes à clé symétrique [106] (dits aussi à clé secrète) sont des algorithmes où la clé de déchiffrement (resp. chiffrement) peut être facilement calculée (ou retrouvée) à partir de la clé réciproque. Généralement les clés de chiffrement et de déchiffrement sont identiques. Par abus de langage, la cryptographie symétrique se réfère à ce cas de figure. Les systèmes basés sur la cryptographie symétrique posent des problèmes de distribution de clé : les protagonistes d'une communication doivent au préalable se mettre d'accord sur la clé à utiliser dans l'échange de données. La sécurité et l'efficacité

d'un algorithme symétrique reposent principalement sur la non divulgation de la clé de chiffrement/-déchiffrement. Celle-ci doit être fournie aux participants dans la confidentialité la plus absolue. La figure 3.2 présente le schéma de cryptographie symétrique dans le cas d'une communication entre une source A, qui utilise une clé K pour le chiffrement d'un message M ($E_k(M)$)et une destination B qui utilise la même clé pour la fonction de déchiffrement D_k.

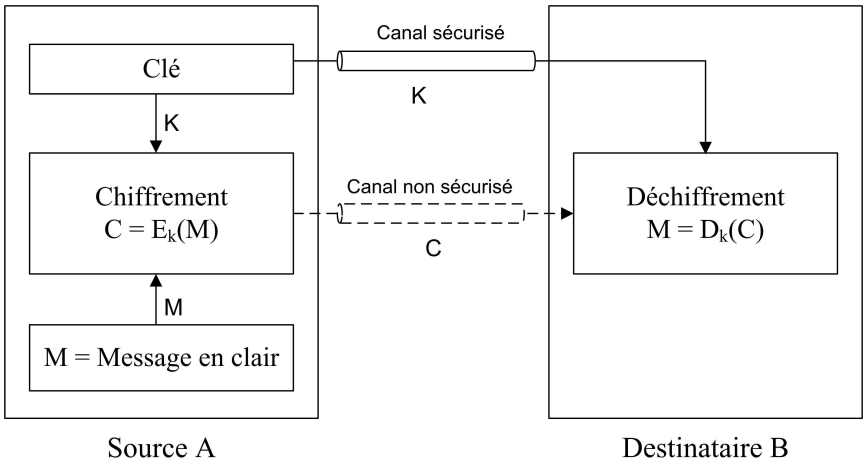


Fig. 3.2 – Schéma de la cryptographie symétrique (d'après [106])

3.3.1.2 La cryptographie asymétrique

La cryptographie à clé asymétrique [106] est conçue de telle manière que la clé de chiffrement soit différente de celle de déchiffrement. Chaque entité possède une paire de clés : l'une privée et l'autre publique. La première doit être gardée secrète par son propriétaire : c'est la clé privée. La deuxième, par contre, pourra être largement diffusée : c'est la clé publique. Bien qu'elles soient différentes, les deux clés sont complémentaires ; elles sont mathématiquement liées. La génération des paires de clés <privée, publique> est basée sur des méthodes modernes utilisant des mathématiques sophistiquées relatives aux nombres premiers. De ce fait, il est extrêmement dur, connaissant la clé publique, de calculer la clé privée. La factorisation des nombres entiers formant les clés nécessite des traitements de calcul considérables et qui pourrait durer plusieurs années. La figure 3.3 présente un schéma de cryptographie dans lequel la clé publique K_{pub} de B (destinataire) est utilisée par A (la source) pour chiffrer les données qui lui sont envoyées (E_{kpub}). En utilisant sa clé privée (K_{prv}), l'entité B peut ainsi déchiffrer le message (D_{kprv}).

La diffusion de la clé publique constitue toutefois un vrai problème. En effet, lorsqu'on souhaite communiquer avec une autre entité il faut s'assurer que : (1) la clé publique que l'on a récupérée est bien celle de l'entité avec laquelle on souhaite communiquer ; (2) la clé est toujours utilisable, c'est-à-dire que la clé privée associée n'a pas été compromise ; (3) l'identité annoncée par le possesseur de la clé est bien la sienne.

Les infrastructures à clés publiques [5] (appelées *Public Key Infrastructure* : PKI) apportent des solutions techniques à ce genre de problèmes. Dans une PKI, les clés publiques sont contenues dans des fichiers appelés certificats. Ces certificats sont créés et signés par un tiers de confiance, l'autorité de certification. Un certificat peut être défini comme étant un document électronique reliant une clé à une identité donnée. Les architectures PKI seront présentées un peu plus loin dans la section 3.5.

3.3.1.3 Les fonctions de hachage

Les fonctions de hachage sont un des concepts fondamentaux de la cryptographie moderne [106]. Une fonction de hachage est une fonction mathématique qui prend en entrée une séquence de données de longueur quelconque et produit en sortie une séquence de taille constante. Le résultat de cette

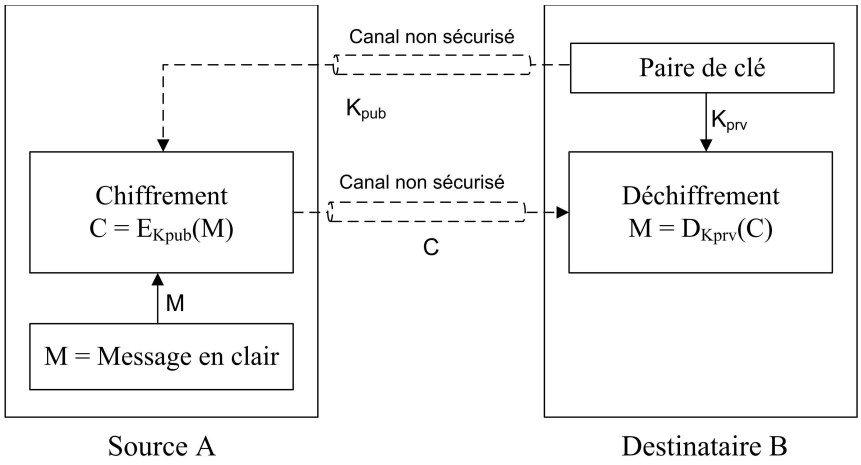


FIG. 3.3 – Schéma de la cryptographie asymétrique (d'après [106])

fonction s'appelle code de hachage ou condensé. Il faut noter que les fonctions de hachage sont définies de telle manière qu'il soit impossible de changer le contenu de la séquence de données en entrée sans altérer le code de hachage de sortie. Autrement dit, il est peu probable, voire quasi-impossible, d'avoir un condensé identique pour deux séquences de données en entrée différentes. Les fonctions de hachage sont des fonctions à sens unique, c'est-à-dire que connaissant le condensé, il n'est pas possible de trouver l'antécédent correspondant (la séquence de données en entrée) qui a conduit à la génération de ce code de hachage.

3.3.2 Schéma symétrique versus Schéma asymétrique

Les schémas de cryptographie symétrique et asymétrique présentent chacun des avantages et des inconvénients. Une étude plus détaillée peut être trouvée dans dans [106] :

1. Avantages de la cryptographie symétrique
   - Les algorithmes à clé symétrique sont relativement simples ce qui permet d'obtenir de bonnes performances lors de leur exécution. Ceci est important pour les systèmes à ressources limitées.
   - La longueur des clés pour ces algorithmes est relativement courte (généralement entre 64 et 256 bits).
2. Inconvénients de la cryptographie symétrique
   - Préalablement à toute communication sécurisée, la clé de chiffrement/déchiffrement doit être remise aux protagonistes selon une procédure sécurisée en garantissant la confidentialité. Ceci n'est pas toujours simple, surtout lorsqu'il s'agit de systèmes et de réseaux ouverts.
   - Pour un système composé de plusieurs nœuds, le nombre de clés mises en jeu peut être considérable. Ce nombre augmente de façon exponentielle avec l'élargissement du système. Pour n participants, et afin d'établir des canaux sécurisés entre toute paire de nœuds, chaque participant doit être en possession de $n-1$ clés. Le nombre total de clés dans le système sécurisé est donc de $n*(n-1)/2$.
3. Avantages de la cryptographie asymétrique
   - À l'inverse de la cryptographie symétrique, seule la clé privée doit être gardée secrète. La clé publique, par contre, peut être largement diffusée.
   - La construction des clés privée et publique est basée sur des propriétés mathématiques relatives aux nombres premiers. Une paire de clés pourra alors rester valable pendant une longue durée sans qu'elle ne soit cassée. La factorisation des clés nécessite une capacité de calcul puissante et un temps très important qui pourrait atteindre des années.
4. Inconvénients de la cryptographie asymétrique

- Les algorithmes à clé publique nécessitent une capacité de traitement importante, ce qui n'est pas raisonnable pour les systèmes à ressources limitées.
- La taille des clés est relativement longue (généralement entre 512 et 2048 bits).

Pour pouvoir bénéficier des avantages des deux schémas de cryptographie, les concepteurs de protocoles de sécurité combinent l'utilisation des algorithmes symétriques et asymétriques. Les clés asymétriques sont généralement utilisées pour établir des clés temporaires, de type symétrique, qui seront employées lors d'une session temporaire.

3.4 Sécurité des communications

Dans un système distribué les communications peuvent être la cible de plusieurs types d'attaque : rejeu, *man in the middle*, usurpation d'identité, etc. Ces différentes attaques peuvent être classées en deux catégories :

- les attaques passives : dans ce type d'attaque, un agresseur écoute une communication entre deux parties sans agir sur les données échangées. Théoriquement, il n'est pas possible de détecter ce genre d'attaques. Cependant il existe des moyens de prévention permettant de rendre non intelligibles les informations sensibles ou à caractère secret.
- les attaques actives : ce type d'attaque consiste à agir sur les données échangée en les modifiant, en les altérant, ou en les supprimant. Contrairement aux attaques passives, les attaques actives sont détectables mais n'admettent pas de moyens de prévention. En général, toute personne possédant un accès au canal de communication peut agir sur les messages échangés. Les attaques actives peuvent être détectées en utilisant par exemple des techniques de contrôle d'intégrité, ou d'authentification.

3.4.1 Les propriétés de sécurité

3.4.1.1 L'authentification

L'authentification est le processus qui consiste à vérifier une identité annoncée par un tiers, et/ou un flux de données. Deux entités qui s'engagent dans une communication doivent s'assurer de l'identité de leur homologue. Les informations reçues à travers un canal de communication doivent également être authentifiées (origine du message, contenu des données, ordre d'émission, etc.).

3.4.1.2 La confidentialité

La confidentialité est le fait de garder le contenu d'une donnée inintelligible aux tiers, à l'exception des entités légitimes. Ainsi, il est possible d'éviter un accès non autorisé aux informations.

Dans la littérature [106], on trouve souvent les termes *secret* et *privacy*, qui sont, dans ce contexte, des synonymes de confidentialité.

3.4.1.3 L'intégrité

Le contrôle d'intégrité permet de se prémunir contre la modification non autorisées des informations échangées (insertion, suppression ou substitution de données).

3.4.1.4 Non répudiation

Si la non-répudiation est assurée, alors il est impossible aux entités de nier des engagements ou des actions précédentes. Par exemple, lorsque l'on utilise des services de non-répudiation, une entité qui a envoyé un message ne peut pas réfuter l'avoir fait.

3.4.2 Mise en œuvre des propriétés de sécurité

Afin de garantir les propriétés de sécurité que nous avons décrites précédemment, plusieurs techniques et procédures existent. On trouve par exemple :

- la signature numérique : une signature numérique est générée en appliquant à la fois une fonction de hachage et un algorithme de cryptographie à clé asymétrique. Le résultat de la fonction de hachage, appliquée sur l'intégralité du message à envoyer (ou sur une des ses parties), est chiffré avec la clé privée associée à la source. La signature numérique, ainsi obtenue, est ajoutée au message avant qu'il soit envoyé. De l'autre côté, le destinataire doit procéder à la vérification du message qu'il reçoit. Il recalcule le code de hachage du message original et décrypte la signature associée en utilisant la clé publique de l'émetteur. En effectuant une comparaison entre les deux valeurs de hachages résultats , le destinataire peut conclure quant à l'intégrité et l'authentification de l'origine. La figure 3.4 présente un schéma pour la génération/vérification des signatures numériques. Pour signer un message, on utilise sa clé privée afin de chiffrer le code de hachage (EKprv(h)) et générer la signature d'un message. Lors de la vérification c'est la clé publique associée qui est utilisée pour déchiffrer (DKpu(s)) et vérifier la signature.

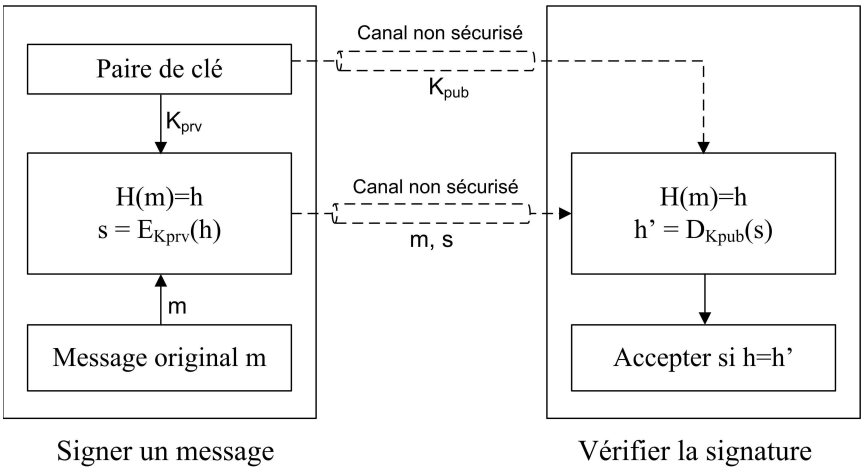


FIG. 3.4 – Schéma de la signature numérique (d'après [106])

- un code d'authentification de message : cette méthode consiste à adjoindre au message un code d'authentification de message, appelé aussi sceau, qui est le résultat d'une fonction à clé secrète. Généralement la clé secrète utilisée est générée selon une procédure bien définie par les deux parties au début de la communication. Il se peut également qu'elle soit partagée en préalable à la communication. Dans tous les cas, il faut que la valeur de cette clé reste confidentielle. Lorsque le destinataire reçoit un message, il utilise la clé pour régénérer le sceau et en effectuant une comparaison peut authentifier la source et s'assurer de l'intégrité de ce message.
- le chiffrement : le chiffrement consiste à appliquer une fonction cryptographique sur les données confidentielles du message. On peut utiliser une fonction à clé symétrique ou à clé asymétrique.
- un fichier certificat : un certificat est un document électronique reliant une clé publique à une identité donnée. L'utilisation de certificats permet d'obtenir la clé authentique d'une partie tierce. Pour cela, il faut tout d'abord vérifier le certificat afin de s'assurer de sa validité (voir section 3.5.2).

3.5 Les infrastructures à clés publiques

3.5.1 Présentation générale

Comme nous l'avons évoqué, les architectures PKI, à travers l'utilisation de certificats, permettent d'apporter des solutions pour obtenir la clé publique authentique d'une entité tierce. Par définition,

une infrastructure à clé publique est un ensemble de moyens matériels, de logiciels, de composants cryptographiques, mis en œuvre par des personnes. Ces moyens sont combinés par des politiques, des pratiques et des procédures requises, qui permettent de créer, gérer, conserver, distribuer et révoquer des certificats basés sur la cryptographie asymétrique (définition empruntée à l'IETF) [5].

Une PKI est fondée sur une autorité de certification qui définit les politiques de gestion, de stockage et de vérification des certificats. Celle-ci représente la partie centrale d'infrastructure à clé publique.

Dans une PKI, chaque entité doit disposer d'une paire de clés, l'une privée et l'autre publique. La clé privée doit être gardée secrète alors que la clé publique est diffusée. Afin de relier une clé publique à son propriétaire légitime, le CA (*Certification Authority*) crée et signe des certificats d'identité, appelés aussi certificats de clé publique. De manière plus précise, un certificat est composé d'une clé publique, des informations d'identité relatives au propriétaire de la clé (nom, statut, rôle, etc.) et d'une signature numérique. L'intégrité des informations contenues dans le certificat est garantie par le fait qu'elles ont été signées par une entité digne de confiance, le CA. Tout utilisateur doit donc avoir confiance en le CA qui émet et gère l'ensemble des certificats mis en jeu.

De plus, chaque certificat possède une période de validité définie par sa date d'émission et sa date d'expiration. Ces informations doivent figurer explicitement dans le certificat. Au delà de la date d'expiration le certificat n'est plus valide. Toutefois, il se peut qu'un certificat donné soit révoqué avant sa date de fin de validité. C'est par exemple le cas lorsque les informations concernant le propriétaire changent (nom, rôle, etc), ou lorsque la clé privée a été compromise, etc. Pour cela, l'autorité de certification doit définir une politique qui établit l'ensemble des règles de vérification des certificats. Dans une infrastructure à clé publique, on pourra adopter une stratégie basée la liste des certificats révoqués (CRL) [123], ou sur un protocole de validation en ligne, de type requête-réponse, tels que SCVP [43], ou OCSP [122]. Dans ces protocoles le client envoie une requête au serveur (OCSP par exemple) qui répond en indiquant l'état du certificat.

De plus, l'autorité de certification doit assurer une fonction de publication des certificats émis en définissant des politiques de stockage et de distribution. Les utilisateurs de la PKI peuvent ainsi avoir accès aux certificats à travers par exemple un annuaire, une application web, ou un service de messagerie. Cette fonction est réalisée par l'autorité de dépôt qui met à la disposition des utilisateurs la liste des certificats reconnus comme valides, ainsi que les certificats révoqués (CRL) [5].

La figure 3.5 illustre, de façon simplifiée le processus de création et de diffusion de certificats dans une infrastructure à clé publique.

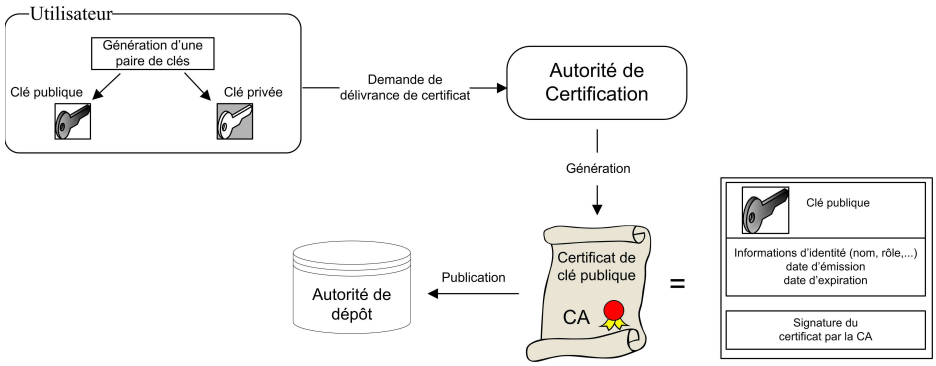


Fig. 3.5 – Création de certificat dans une PKI

Un utilisateur qui souhaite effectuer une transaction avec un tiers doit récupérer le certificat de ce dernier, soit directement soit indirectement à travers l'autorité de dépôt, afin de pouvoir initier une communication sécurisée. Il est nécessaire lors de ce processus de vérifier la validité du certificat de la partie tierce avant de l'utiliser.

3.5.2 Processus de vérification des certificats

En plus du CA principal, il est possible de trouver dans une même architecture PKI d'autres autorités de certification. Celles-ci sont appelées des autorités subordonnées. Le CA principal utilise sa clé privée pour signer les certificats (et en particulier les certificats des autorités subordonnées) qui sont alors considérés comme valides. De la même façon, une autorité subordonnée, possédant un certificat signé par le CA principal, peut signer des certificats, éventuellement pour d'autres autorités subordonnées. Le processus de contrôle et de validation des certificats intermédiaires est appelé suivi de chemin de certification ou de chaîne de certification.

Le deuxième aspect à considérer lors de la vérification des certificats est leur éventuelle révocation. Un certificat peut en effet être révoqué avant sa date de fin de validité. Le processus de validation consiste donc non seulement à vérifier les différents champs du certificat (signature, date d'expiration, etc.) mais aussi à s'assurer qu'il n'a pas été révoqué. Tout système basé sur l'utilisation de certificats doit donc fournir une procédure sûre et efficace pour vérifier ces point.

D'une façon générale ce processus suit plusieurs étapes que nous présentons ci-dessous (une étude détaillée peut être trouvée dans [139]) :

- une vérification syntaxique. On s'assure que le certificat est bien structuré et qu'il respecte la spécification syntaxique (champs obligatoires, extensions, etc.) définie dans la norme appropriée.
- un suivi de chemin de certification (s'il en existe un). Il s'agit de l'étape de vérification la plus délicate. Il est nécessaire de récupérer les certificats de parents successifs jusqu'à obtenir le certificat du CA principal, ou d'un des subordonnés dignes de confiance. Ensuite, chaque certificat ainsi obtenu doit être vérifié (vérification syntaxique, cryptographique et vérification d'état).
- une vérification cryptographique. Elle consiste à s'assurer que les informations contenues dans le certificat n'ont pas été modifiées depuis la signature de ce dernier. La clé utilisée pour la vérification est celle contenue dans le certificat hiérarchiquement supérieur, c'est-à-dire la clé de l'autorité qui a signé le certificat.
- une vérification sémantique. Il faut vérifier que le certificat n'a pas expiré (la date de fin de validité doit être postérieure à la date courante).
- une vérification d'état. Il faut vérifier que le certificat n'a pas été révoqué.

Une fois que toutes ces vérifications sont passées, le certificat est considéré valide et peut être utilisé.

3.6 Le modèle symétrique dans GlobalPlatform

En se basant sur la cryptographie symétrique, GlobalPlatform définit deux protocoles de communication pour l'établissement d'un canal sécurisé entre la carte et l'entité extérieure, SCP01 et SCP02. Les propriétés qu'un tel canal permet d'assurer sont :

- l'authentification de l'homologue : avant d'accéder aux services de la carte, l'entité extérieure et la carte commencent par s'authentifier. Pour cela, les deux protagonistes doivent prouver qu'ils partagent un même secret.
- la confidentialité des données : seul le destinataire doit pouvoir déchiffrer le contenu des messages qui lui sont transmis.
- l'authentification de la source : le destinataire d'un message doit pouvoir vérifier l'authenticité de son origine. Si un intrus se fait passer pour quelqu'un d'autre, son acte pourra être détectée.
- l'intégrité des messages : le destinataire doit s'assurer que les messages n'ont pas été altérés et qu'ils sont reçus dans l'ordre de leur émission. L'intégrité permet également de détecter les attaques de type rejeu grâce à un chaînage établi entre les messages.

3.6.1 Préalable à l'établissement d'un canal sécurisé

Conformément aux spécifications GlobalPlatform, toute application installée sur la carte doit être liée à un domaine de sécurité qui lui fournit les procédures et les fonctions de sécurité requises (authentifier une entité extérieure, chiffrer/déchiffrer les messages, etc.). Le domaine de sécurité dispose d'un jeu de clés statiques. Si l'on suit l'implémentation de base de GlobalPlatform, ces clés sont des clés symétriques. Elles sont partagées avec l'utilisateur, ce qui permet aux deux entités de s'authentifier : chacun d'eux doit prouver qu'il dispose de cette information secrète. Ces clés sont aussi employées pour générer de nouvelles clés elles mêmes symétriques, dites clés de session. Les nouvelles clés ne sont valides que pour la session en cours et sont utilisées afin d'assurer la confidentialité des données échangées et l'intégrité des messages transmis.

3.6.2 L'authentification mutuelle

Le schéma d'authentification mutuelle défini dans les spécifications GlobalPlatform pour le modèle symétrique est basé sur des clés statiques, qui doivent être connues par la carte et l'entité extérieure. Chaque protagoniste vérifie l'identité de son homologue en s'assurant qu'il partage avec lui le même jeu de clés. L'authentification mutuelle est réalisée dans le but d'initialiser un canal sécurisé et de fournir la garantie à la carte et à l'entité extérieure qu'ils communiquent chacun avec une entité authentique. Dans le cas où les deux interlocuteurs ne parviennent pas à s'authentifier, le canal sécurisé ne pourra pas être établi.

Les étapes d'authentification mutuelle sont présentées à la figure 3.6. Comme nous l'avons déjà évoqué, l'application carte fait appel aux services de son domaine de sécurité qui gère l'établissement d'un canal sécurisé : authentification de l'entité extérieure, chiffrement/déchiffrement des messages, etc.

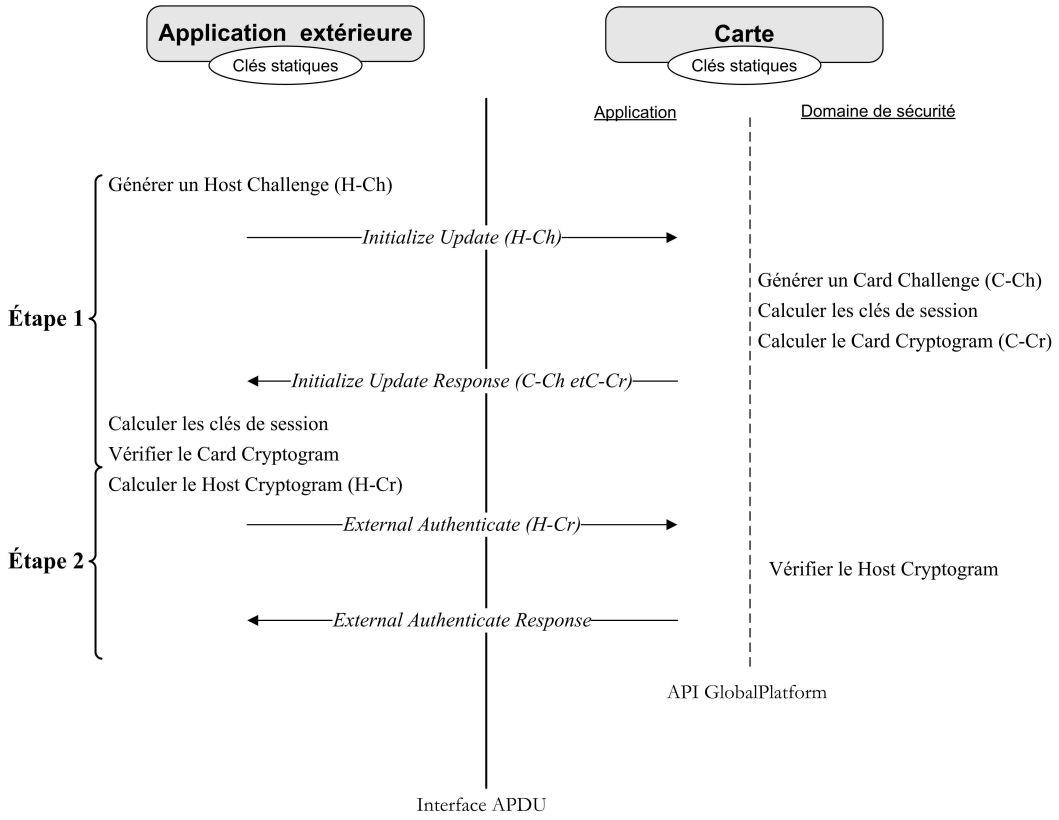


Fig. 3.6 – Étapes d'authentification mutuelle (d'après GlobalPlatform [65])

Étape 1 : Authentification de la carte

- La première étape de l'authentification mutuelle est l'envoi d'une commande APDU contenant un nombre aléatoire appelé *Host Challenge*. Cette commande, envoyée par l'entité distante à la carte, est connue sous le nom de *Initialize Update Command*.
- À la réception de cette commande, la carte génère un deuxième nombre aléatoire appelé *Card Challenge*. À partir des clés statiques partagées, elle crée selon la procédure définie dans les spécifications GlobalPlatform [66] de nouvelles clés, appelées clés de session .
  Ensuite, la carte produit un cryptogramme appelé *Card Cryptogram* en utilisant des clés de session et des nombres aléatoires (le *Card Challenge* et le *Host Challenge*) selon la méthode décrite dans GlobalPlatform [66]. Le *Card Cryptogram*, accompagné du *Card Challenge* sont renvoyés à l'entité distante dans une réponse APDU appelée *Initialize Update Response*.
- Ainsi, l'entité extérieure dispose de toutes les données et informations que la carte a employées pour produire son *Card Cryptogram*. Connaissant la procédure utilisée pour générer les clés de session ainsi que celle employée pour calculer le *Card Cryptogram*, l'entité distante est donc capable de produire de la même façon un *Card Cryptogram*. Enfin, en effectuant une comparaison entre le *Card Cryptogram* reçu et la valeur qu'elle a générée, elle pourra authentifier la carte si les deux valeurs correspondent.

Étape 2 : Authentification de l'entité extérieure

- Dans cette deuxième étape, l'entité extérieure génère un *Host Cryptogram*, selon une méthode semblable à celle employée par la carte pour générer le *Card Cryptogram*, et l' envoie à la carte dans une commande APDU appelée *External Authenticate*. Notons que c'est dans l'entête de cette commande que l'utilisateur choisit le niveau de sécurité (voir section 3.6.3) à appliquer dans le reste de la communication.
- Après réception de cette commande, la carte génère de la même façon un *Host Cryptogram* et en effectuant une comparaison avec la valeur reçue, elle peut à son tour authentifier l'entité distante.
- Pour finir, la carte renvoie une réponse indiquant la réussite ou l'échec de l'authentification de son correspondant (l'entité extérieure), et dans l'hypothèse positive, un canal se trouve établi.

À l'issue d'une authentification mutuelle réussie, les deux entités authentifiées partagent des clés de session qui sont utilisées pour réaliser une communication sécurisée. Ces clés sont des dérivées des clés statiques (appelées aussi clés basiques) et elles dépendent de paramètres propres à la session en cours (par exemple les *challenges*) Elles sont utilisées par la suite afin d'assurer la confidentialité des données et l'intégrité des messages de la session courante.

3.6.3 Niveau de sécurité de la communication

Les spécifications GlobalPlatform définissent différents niveaux pour le canal sécurisé selon que l'on cherche à garantir la confidentialité des commandes, leur intégrité, l'intégrité des réponses, ou une combinaison des trois. Pour récapituler, les différents niveaux de sécurité sont les suivants :

- niveau authentification : il permet uniquement l'authentification des interlocuteurs selon la procédure d'authentification mutuelle décrite précédemment. Aucune sécurisation n'est effectuée ni pour les commandes, ni pour les réponses APDU, autrement dit les messages sont envoyés en clair.
- niveau authentification et intégrité : en plus de l'authentification, l'intégrité des messages APDU est assurée. L'intégrité pourra être mise en œuvre pour les commandes, les réponses ou les deux.
- niveau authentification, intégrité et confidentialité : la confidentialité des données (le champs données dans la commande) est garantie en plus de l'intégrité des commandes. C'est le niveau de sécurité maximum.

Dans les spécifications GlobalPlatform, aucune forme de sécurisation ne peut être appliquée pour les réponses APDU lorsque le protocole SCP01 est utilisé. Pour SCP02, seule l'intégrité des réponses peut être mise en œuvre. Le tableau 3.1 présente les différentes combinaisons possibles pour l'établissement de canaux sécurisés selon que l'on utilise SCP01 ou SCP02.

	authentification mutuelle	Commandes		Réponses	
		confidentialité	intégrité	intégrité	confidentialité
SCP01	√			Non supportée	Non supportée
	√		√		
	√	√	√		
SCP02	√				Non supportée
	√		√		
	√	√	√		
	√			√	
	√		√	√	
	√	√	√	√	

Tab. 3.1 – Les différents niveaux de sécurité proposés par GlobalPlatform

3.6.4 Intégrité des messages

L'intégrité est la propriété assure que ni l'ordre des messages, ni leur contenu n'ont été altérés. Elle concerne les commandes APDU aussi bien que les réponses. Pour les commandes, l'intégrité est assurée en générant un code d'authentification de message (MAC). Afin d'éviter les attaques de type rejeu, un chaînage est créé entre les commandes. Pour cela, la génération du MAC d'une commande (le C-MAC)fait intervenir le C-MAC précédent, comme le montre la figure 3.7. Le C-MAC est généré en appliquant l'algorithme DES avec une des clés de session établies lors de la phase d'authentification mutuelle. La carte, à la réception d'un message contenant un MAC, utilise la même procédure que celle employée par l'entité distante pour produire le MAC correspondant à la commande reçue. Puis en comparant le MAC calculé au MAC reçu s'assure de l'intégrité de la commande APDU.

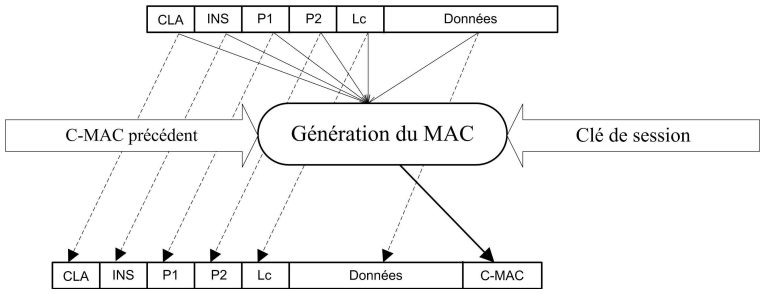


Fig. 3.7 – Procédure de génération du MAC pour une commande APDU

L'intégrité des réponses lorsqu'elle est souhaitée, est également réalisée à travers la génération d'un code d'authentification. La génération de ce MAC fait intervenir la réponse APDU, ainsi que la commande APDU correspondante, et une des clés de session générée dans la phase d'authentification mutuelle, comme illustré à la figure 3.8. Le MAC de la réponse APDU (R-MAC) est ajouté à la fin du message à envoyer. Dans l'implémentation de référence de GlobalPlatform, la génération du R-MAC est réalisée en utilisant la fonction cryptographique DES.

3.6.5 Confidentialité des messages

La confidentialité des messages est la garantie que des données puissent être transmises à travers un réseau ouvert sans crainte qu'elles ne soient déchiffrées, même dans le cas où elles seraient interceptées par un attaquant. La confidentialité est réalisée en appliquant la fonction cryptographique DES sur le champs de données de la commande APDU à transmettre à la carte en utilisant une des clés générées dans la phase d'authentification mutuelle.

Pour résumer, la structure d'une commande APDU sécurisée selon le niveau maximum (intégrité et confidentialité) est illustrée par la figure 3.9.

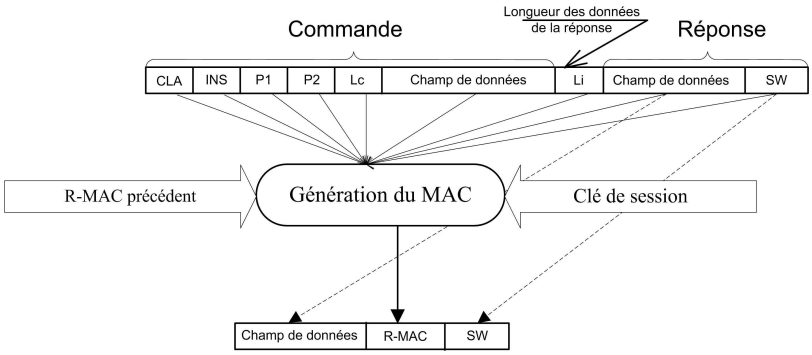


FIG. 3.8 – Procédure de génération du MAC pour une réponse APDU

Entête CLA, INS, P1, P2, Lc	Données chiffrées DES (données en clair)	MAC fonction (MAC de la commande précédente)

FIG. 3.9 – Format d'une commande sécurisée

Le format d'une réponse APDU sécurisée est donné dans la figure 3.10. Rappelons que la confidentialité des réponses n'est supportée ni par SCP01, ni par SCP02 et que l'intégrité ne peut être mise en œuvre qu'avec SCP02.

Données en clair	MAC fonction (commande APDU, MAC de la réponse précédente)

FIG. 3.10 – Format d'une réponse sécurisée

3.7 Le modèle asymétrique dans *GlobalPlatform*

Les spécifications GlobalPlatform définissent un troisième protocole de communication, le SCP10, basé sur l'utilisation de la cryptographie asymétrique et des certificats s'appuyant sur une infrastructure à clés publiques (PKI). L'utilisation des clés asymétriques se limite à l'initialisation du canal sécurisé, c'est-à-dire à la phase d'authentification mutuelle. Une fois que les deux parties se sont authentifiées, de nouvelles clés symétriques (les clés de session) sont générées pour sécuriser les messages pour le reste de la communication.

Comme dans l'approche symétrique, l'application carte fait appel aux services de son domaine de sécurité qui se charge d'authentifier l'entité extérieure et de gérer le canal sécurisé. Le protocole de communication SCP10 permet d'assurer les trois aspects de sécurité suivants :

- authentification : la carte, à travers le domaine de sécurité de l'application, authentifie l'entité distante. Réciproquement, l'application extérieure authentifie le domaine de sécurité relatif à l'application carte. Cette authentification mutuelle consiste à vérifier la validité du certificat de son homologue, ce qui permet par ailleurs de récupérer sa clé publique authentique.
- intégrité : la carte et l'entité extérieure vérifient que les données reçues proviennent bien de l'interlocuteur authentifié. Il faut également s'assurer que les messages sont reçus dans l'ordre de leur émission et qu'ils n'ont pas été altérés lors de la transmission.
- confidentialité : les données échangées ne sont pas dévoilées à des entités non autorisées.

Afin d'obtenir la clé publique de son homologue, le domaine de sécurité de l'application carte doit vérifier le certificat fourni par l'entité extérieure. Ce certificat doit être délivré soit par l'autorité de

confiance principale pour l'authentification externe, TP_EX[1] (*Trust Point for External Authentication*), comme le montre la figure 3.11, soit par une autorité subordonnée au TP_EX. Le domaine de sécurité doit donc disposer au moins de la clé publique du TP_EX. De plus, lorsque le certificat est délivré par une autorité subordonnée, l'entité de confiance doit fournir à la carte tous les certificats intermédiaires (ou au moins leur identifiant s'ils sont déjà chargés sur la carte) pour permettre la vérification de la chaîne de certification correspondante et donc de valider le certificat de l'entité extérieure.

De son côté, l'entité extérieure doit établir la validité de la clé publique du domaine de sécurité en vérifiant le certificat qui a été délivré par l'autorité principale du domaine de sécurité, TP_IN (*Trust Point for Internal Authentication*) (figure 3.11). Il se peut également que le certificat du domaine de sécurité ait été délivré par une autorité subordonnée. L'entité extérieure doit alors récupérer tous les certificats intermédiaires et vérifier la chaîne de certification puis valider le certificat de la carte.

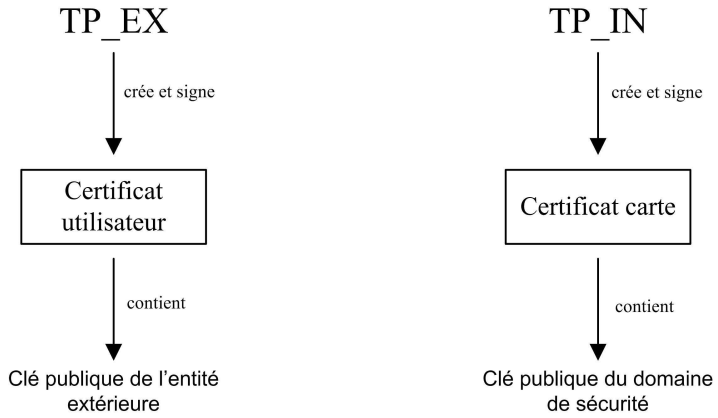


Fig. 3.11 – Un modèle pour la délivrance de certificats dans GlobalPlatform

L'établissement d'un canal sécurisé, selon le protocole SCP10 consiste en :
- la vérification de certificat. Le domaine de sécurité de la carte et l'application extérieure doivent vérifier la chaîne de certification de leur correspondant afin d'établir la validité de la clé publique associée (voir section 3.7.1).
- l'authentification et l'identification. Le domaine de sécurité de la carte, ainsi que l'application extérieure valident l'authenticité et l'identité de leur correspondant via un mécanisme de type *challenge-response*. Cette phase est décrite dans la section 3.7.2.

3.7.1 Vérification des certificats

Dans cette première phase, la carte et l'entité extérieure valident la clé publique de leur homologue. Pour cela chacune des deux parties dispose d'un certificat qui a été délivré par une autorité de certification (TP_EX ou TP_IN), ou par une de leurs autorités subordonnées. Pour pouvoir vérifier le certificat de l'autre entité, chacun des deux participants dispose des clés publiques des autorités de certification. Plusieurs scénarios peuvent se présenter lors de la vérification de la chaîne de certificats, mais cette procédure de vérification obéit au schéma global illustré à la figure 3.12.

Durant cette phase de vérification, plusieurs types de commandes sont employés. Pour échanger leurs certificats et établir la chaîne de certification, les deux participants (carte et entité extérieure) utilisent les commandes *Get Data [certificate]* et *Perform Security operation [verify certificate]*. Pour initialiser cette phase, l'entité extérieure peut éventuellement envoyer une commande de type *Manage Security Environment* (commande optionnelle), ce qui permet de spécifier les paramètres à utiliser durant le protocole SCP10.

[1] C'est le terme utilisé dans les spécifications GlobalPlatform et qui correspond à une autorité de certification.

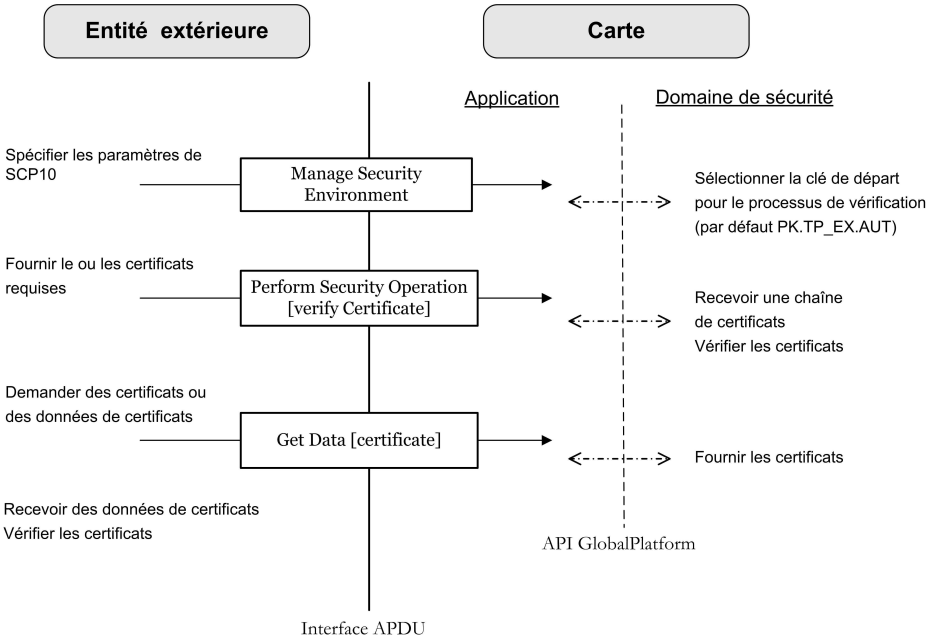


FIG. 3.12 – L'étape de vérification dans SCP10 (d'après GlobalPlatform [66])

Lorsque la carte reçoit une commande de type *Manage Security Environment*, le niveau de sécurité est initialisé et toute autre session sur le même canal logique est terminée. Cette commande contient des valeurs fixant les paramètres du protocole pour l'établissement du canal sécurisé. Lorsque cette commande est omise, le domaine de sécurité utilise les valeurs par défaut. Par exemple, dans ce cas, la clé publique par défaut (la clé publique du TP_EX) est utilisée comme la clé racine pour la vérification d'une chaîne de certificats. Il est possible que la carte reçoive une suite de commandes de type *Perform Security operation [verify certificate]*, ceci signifie l'existence d'une chaîne de certificats.

À l'issue de cette phase, les deux entités disposent de la clé publique de l'autre entité, clé qui sera utilisée dans la suite du processus d'établissement du canal sécurisé.

3.7.2 Authentification et identification

Dans la deuxième phase de l'établissement du canal sécurisé et en se basant sur un mécanisme de *challenges-responses*, les deux entités peuvent s'authentifier. Les messages échangés durant cette phase sont signés avec la clé privée de chaque protagoniste. À la fin de cette phase, les deux parties se mettent d'accord sur un jeu de clés symétriques à utiliser dans la suite de la communication.

GlobalPlatform définit plusieurs types de messages (commandes et réponses) qui sont utilisés durant cette phase tels que la commande *Get Challenge*, les commandes *Internal* et *External Authenticate* comme le montre la figure 3.13.

La commande *Perform Security Operation [decipher]* permet d'envoyer les clés de session à utiliser. La commande *Get Challenge* permet à l'entité extérieure de demander à la carte un *Card Challenge* qui sera envoyé dans une réponse de même type. Ce *Challenge* sera utilisé par la suite dans la phase d'authentification. Les messages *Internal Authenticate* et *External Authenticate* permettent aux deux parties de s'authentifier en utilisant un jeu de messages chiffrés et signés.

À l'issue de cette étape, les deux parties partagent des clés de session symétriques à utiliser dans le reste de la communication.

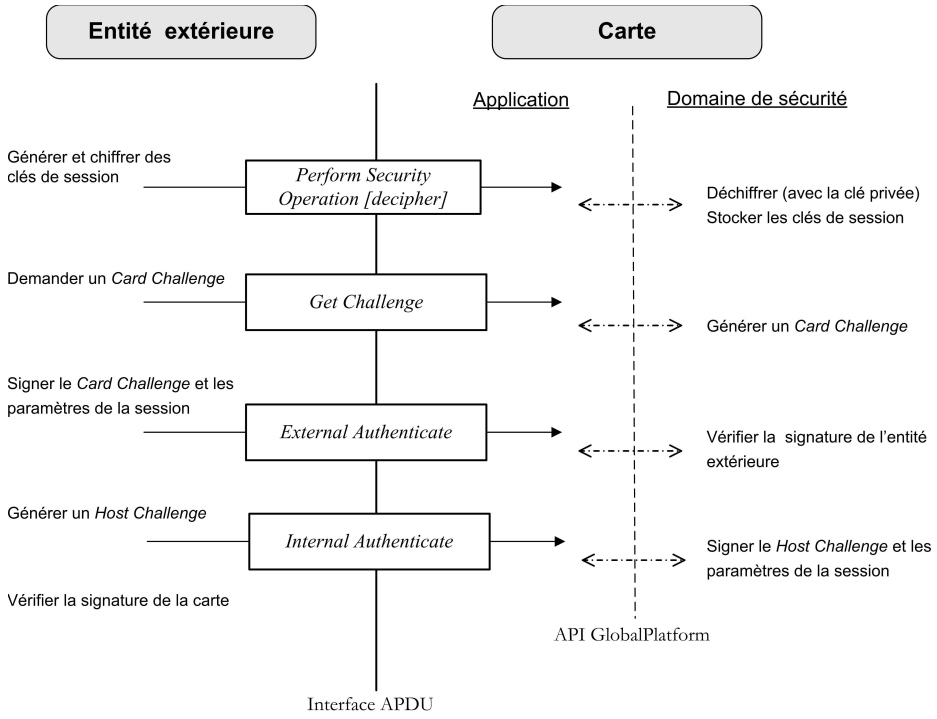


Fig. 3.13 – L'étape d'authentification dans SCP10 (d'après GlobalPlatform [66])

3.8 Conclusion

Comme nous venons de le décrire à travers ce chapitre, GlobalPlatform propose un certain nombre de fonctionnalités qui permettent de résoudre les problèmes de sécurité posés par les cartes ouvertes. Ce standard offre une plate-forme efficace qui fournit un niveau de sécurité optimal. Elle a déjà été déployée sur les cartes SIM et les cartes bancaires. Plusieurs cartes multi-applicatives intègrent une implémentation des spécifications GlobalPlatform.

Chapitre 4

Une architecture pour les grilles de cartes à puce

Sommaire

4.1 Introduction

Bien qu'il soit relativement récent, le concept de grille informatique est déjà largement développé et utilisé dans la communauté informatique. Il a suscité des travaux au sein de nombreuses équipes de recherche et plusieurs environnements logiciels pour les grilles ont été développés [63, 71, 74, 144]. Comme première définition, on peut dire que les modèles à base de grille sont des architectures distribuées permettant de mettre à disposition des ressources matérielles au service d'un individu ou une entreprise.

La première partie de ce chapitre sera consacrée à la présentation des grilles informatiques et des problèmes de sécurité qu'elles posent. Dans la deuxième partie, nous nous intéressons en particulier

aux grilles de cartes à puce, pour les quelles nous préciserons une architecture matérielle et définissons une architecture logicielle. Nous étudierons ensuite les problèmes liés à la mise en place d'une telle grille et leurs solutions.

4.2 Les grilles informatiques

Par analogie avec un réseau d'électricité, la notion de grille de calcul (*computational grid*) est définie comme étant une infrastructure matérielle et logicielle fournissant un accès fiable, cohérent, très ouvert et de faible coût à une grande capacité de traitement et de calcul [58]. Cette définition propre aux grilles de calcul peut être étendue aux grilles informatiques en général. Lorsqu'il s'agit de capacité de stockage, l'infrastructure grille fournit à l'utilisateur les moyens pour disposer d'un espace de stockage important et pour y accéder.

Cette définition met l'accent sur les aspects matériel et logiciel qui sont les deux piliers d'une grille informatique. Au niveau matériel, une grille se présente comme une interconnexion de plusieurs sortes de ressources (telles que des stations de travail, des serveurs, des supercalculateurs, des grappes, etc.) reliées à travers plusieurs types de réseaux. D'un point de vue logiciel, une grille fournit les procédures et les mécanismes qui permettent un accès facile et efficace aux différentes ressources disponibles tout en offrant à l'utilisateur une abstraction vis à vis de l'hétérogénéité et de la dispersion des différents matériels la constituant.

4.2.1 Différents types de grilles

Selon le type de l'application, les grilles peuvent être répertoriées en trois classes [93, 120, 126] :

Les grilles de données : ce type de grille permet de gérer une grande quantité de données (dont la taille peut atteindre le *terabytes* ou même le *petabytes*) distribuée sur plusieurs sites. La fonction de gestion consiste à assurer des services de partage, de publication et de stockage des données considérées. Les grilles de données sont principalement utilisées dans le cadre des grands projets où les scientifiques et les ingénieurs ont besoin de partager des résultats, des informations statistiques, etc.

Les grilles d'information : les grilles d'information (appelées aussi grilles de connaissance) permettent aux utilisateurs d'accéder à toutes sortes de connaissances sans pour autant se préoccuper de la façon dont ces informations sont stockées, ni comment elles sont représentées, ni de la manière dont elles ont été récupérées. *Internet* est sans doute l'exemple le plus significatif de cette classe de grille. L'utilisateur a accès à l'information à travers les protocoles *http*, *ftp*, etc.

Les grilles de calcul : une grille de calcul permet de relier des ressources informatiques (en particulier des processeurs) afin d'obtenir une puissance de calcul importante. L'accès aux différents nœuds nécessite un *middleware* permettant de gérer l'ordonnancement des processus, la réservation des ressources, les communications, etc. Parmi ces middlewares citons Globus [71, 56, 57], Legion [110, 28, 75], etc. Des standards pour la définition des grilles ont été définis récemment, notamment le *Open Grid Services Architecture* OGSA [55, 63].

Récemment, est apparu le concept de *cloud computing* [105] comme évolution des technologies des grilles informatiques. Le concept du *cloud computing* consiste à utiliser les capacités de calcul et de stockage des ordinateurs et des serveurs répartis dans le monde entier liés par Internet. À l'mage du réseau électrique, les capacités mises à disposition sont fournies par des compagnies spécialisées. Une couche logicielle de gestion des ressources est mise œuvre, permettant ainsi de rendre disponibles les ressources matérielles. Le consommateur (l'utilisateur) peut accéder facilement aux services disponibles à travers une application simple (généralement un navigateur web).

4.2.2 Quelques plate-formes logicielles pour les grilles

Il existe de nombreux projets liés aux grilles, aussi bien académiques qu'industriels. Dans cette section, nous présentons trois de ces infrastructures : Globus, Legion et OGSA.

Globus

Globus [56, 57, 71] est une plate-forme logicielle fournissant les services de base pour les infrastructures de type grille. Elle fournit une architecture multi-composants, chaque composant offrant chacun un service déterminé. Nous présentons ci-dessous quelques services fondamentaux.

- La sécurité : Les mécanismes de sécurité sont mises en place grâce à une infrastructure de sécurité [59] appelée GSI (*Grid Security Infrastructure*), permettant d'offrir les procédures d'autorisation et d'authentification ce qui permet de gérer et de contrôler les droits d'accès.
- Allocation de ressources : Ce service est fourni par un module appelé GRAM (*Globus Resource Allocation Monitor*). Ce composant permet d'assurer l'allocation des ressources pour plusieurs processus ce qui permet d'exécuter en paralléle différents programmes et applications sur l'ensembles des ressources de la grille.
- Découverte de la topologie : Cette fonction est basée sur le module MDS (*Metacomputing Directory Service*) [38] [37] responsable de la découverte des ressources disponibles. Il s'agit d' une sorte d'annuaire qui permet d'enregistrer les ressources formant de la grille.

D'autres composants sont aussi définis dans Globus, par exemple la gestion des communications. L'ensemble de ces différents modules forment le *toolkit*, la boîte à outils, Globus.

Legion

Legion [28, 110, 75] est un intergiciel pour les architectures de type grille basé sur le paradigme objet [76]. Cet environnement fournit à ses utilisateurs un ensemble de services fondamentaux. Ces services concernent principalement le stockage, la gestion des processus, les communications, la gestion des ressources ainsi que la sécurité. Legion définit aussi un espace de nommage appelé *context space*, qui permet à un utilisateur de découvrir facilement l'ensemble des ressources disponibles dans la grille. Ces différents services sont mis en place à travers un environnement intégré et cohérent, où la notion d'objet est l'élément fondamental.

L'architecture Legion définit aussi un modèle entièrement basé sur les techniques de la programmation orientée objet ; tout est donc représenté par un objet (les fichiers, les processus, les connexions, les utilisateurs, les ressources, etc.). Le système complet constitue une machine virtuelle très puissante [75].

OGSA

On ne peut pas clôturer cette section sans parler de OGSA [63] (*Open Grid Services Architecture*). Ce projet, qui a été lancé en 1998, a pour objectif de définir un standard pour les architectures de type grille. OGSA est mis en place à travers le GGF[1], un forum regroupant plusieurs institutions académiques et des industriels. OGSA définit une architecture orientée services, se basant sur la notion de *Web Service* [20]. Le standard OGSA représente une évolution pour les grilles aux quelles il adapte les concepts et les technologies des *Web Services*, notamment les standards WSDL[2] et SOAP[3]. L'architecture OGSA étend la notion de service pour toutes les entités de la grille (ressources matérielles, processus, programmes, etc.) ; on parle de *Grid Service*.

On notera qu'OGSA est une spécification, un standard pour les grilles, et non pas une implémentation. L'alliance Globus est une des premières à adopter les spécifications OGSA [55]. À partir de sa version 3 (juin 2003) la boîte à outils Globus fournit une implémentation basée sur les services grilles conformément aux spécifications OGSA.

[1] Global Grid Forum
[2] Web Service Definition Language
[3] Simple Object Access Protocol

4.3 Les problèmes de sécurité pour les infrastructures de type grille

Les grilles informatiques, et en particulier les grilles de calcul, impliquent d'installer et d'exécuter une application, provenant de tiers, sur une machine appartenant à une autre entité. Le client qui reçoit un code doit donc s'assurer que l'exécution de celui-ci n'affectera pas sa machine d'une manière ou d'une autre (atteinte aux ressources, vol ou corruption de données, etc.). Le propriétaire du code pour sa part cherche à protéger son application d'attaques extérieures qui proviendraient essentiellement d'un accès physique aux ressources matérielles sur lesquelles son programme s'exécute. Nous présentons dans ce qui suit les principaux problèmes liés à la sécurité du code et du matériel. Ils sont connus et déjà largement décrits dans la littérature [82, 103, 128]. Nous les reformulons ici en synthétisant différentes sources.

4.3.1 Protection du matériel envers un code malicieux

Mettre son matériel à disposition de tiers pour y exécuter des applications représente un danger éventuel pour le propriétaire du matériel. En effet, il est possible que le programme à installer cache un code malicieux tel qu'un virus ou un cheval de Troie. Pour se protéger contre ces menaces, il est possible d'utiliser la technique de *Sandboxing* [103]. Cette solution consiste à exécuter le code mobile dans un environnement restreint appelé *sandbox* (bac à sable). L'application n'aura alors accès qu'aux services définis par le propriétaire du matériel. L'inconvénient principal de cette technique est que les applications qui s'exécutent dans des environnements réduits sont rarement utiles. Une autre solution consiste à définir des droits d'accès et des permissions pour l'exécution d'un code mobile [72] comme c'est le cas par exemple pour Java à travers les politiques de sécurité [94] (*policy File*) introduites depuis la JDK 1.2. D'autres solutions, décrites dans [82, 103, 128], basées sur la signature du code ou des modèles de vérification peuvent aussi être utilisées.

Afin de pouvoir vérifier l'innocuité de l'application, le propriétaire de matériel préfère généralement que l'application soit en *open source*, ou au moins qu'il soit possible d'avoir accès à son code. Ainsi, il pourra examiner lui-même ce code ou faire appel à une entité de confiance qui procédera à sa vérification et s'assurera de son innocuité. L'inconvénient majeur de cette solution est qu'elle met en péril la confidentialité du code.

4.3.2 Confidentialité du code

Pour diverses raisons, le propriétaire d'une application peut ne pas vouloir dévoiler les détails d'implémentation de celle-ci. Ceci est le cas pour des domaines critiques comme ceux traitant des données privées et/ou sensibles, notamment pour des applications de type fouille de données ou de type militaire. Une confidentialité parfaite [129] consiste en une exécution durant laquelle le propriétaire du matériel n'arrive pas (ou au moins aura du mal) à comprendre le fonctionnement de l'application et à lire le code et les données manipulées. Ceci permet de définir une boîte noire idéale. Plusieurs techniques sont utilisées pour assurer la confidentialité de l'application et qui sont décrites dans [129] :

- chiffrement de l'exécution : en utilisant des procédures de chiffrement, l'application subit des transformations dans le but de dissimuler le code source. Ainsi, lorsqu'elle est exécutée, il n'est pas possible de découvrir ou de comprendre les vrais traitements qui sont effectués. Des techniques basées sur les polynômes et les fonctions rationnelles peuvent être employées afin de dissimuler le code orignal. Cependant, ces approches sont parfois inefficaces et présentent des fonctionnalités limitées vus la complexité des transformations effectuées et le coût de traitement qui en découle.
- *obfuscation* du code : dans cette technique, le code de l'application est réécrit (en utilisant par exemple des méthodes heuristiques) de façon à ce qu'il soit assez flou et ambigu pour ne plus pouvoir être ni lu ni compris. La nouvelle version du programme doit cependant être équivalente au code original et réaliser les mêmes fonctionnalités. C'est la forme modifiée de l'application qui est envoyée aux différents nœuds de la grille pour être exécutée.

– boîte noire limitée dans le temps : cette approche consiste à diminuer les contraintes de sécurité. Le propriétaire du code cherche à établir une boîte noire restreinte (c'est-à-dire qui sera limitée dans le temps) par mise en œuvre de plusieurs techniques. Il s'agit de protéger l'application, pendant une durée fixée (ce qui limite les attaques potentielles), contre la lecture (compréhension du source) et/ou la modification du code et des données manipulées. Pour le propriétaire du code, les attaques qui pourront être menées au delà de cette période n'auront aucun effet même si elles aboutissent à un succès.

4.3.3 Intégrité de l'exécution

Le propriétaire d'une application souhaite que le résultat qui lui est fourni par son application corresponde bien à une exécution intègre de son code, comme si cette application s'exécutait sur sa propre machine. Pour une raison ou une autre, le propriétaire de matériel pourrait agir sur le comportement de l'application et altérer son fonctionnement. Il lui est possible de perturber l'exécution des programmes puisqu'il dispose d'un accès physique au support matériel. Il lui aussi est facile d'altérer les résultats de l'application en agissant sur les messages renvoyés.

Pour remédier à ce problème, il est possible d'utiliser les techniques de la tolérance aux fautes. Il suffit de répéter plusieurs exécutions d'un même code sur plusieurs nœuds distincts, et d'effectuer une vérification des résultats lors de la réception. Une autre solution consiste à utiliser une validation formelle pour vérifier l'intégrité de l'exécution. Il est possible par exemple d'utiliser un système de preuve holographique afin de s'assurer que l'exécution de l'application est conforme aux instructions du code mobile [148].

4.3.4 Sécurité des échanges

La sécurité des échanges est un problème commun à tous les systèmes distribués. Ce sont toujours les mêmes propriétés (présentées dans la section 3.4.1) que l'on cherche à assurer, à savoir la confidentialité des données échangées, l'intégrité des messages, l'authentification de la source d'un message et la non-répudiation des messages. Ces propriétés sont mises œuvre en tout ou partie selon la nature et le contexte de l'application. Pour les architectures de type grille, la sécurité des communications est soit fournie comme service de base par l'infrastructure logicielle, soit elle est gérée dans la couche applicative.

4.3.5 Notre solution

Nous considérons que l'utilisation de la carte à puce constitue une solution aux problèmes évoqués précédemment. En ce qui concerne la sécurité du support d'exécution, le système d'exploitation de la carte fournit des mécanismes permettant une séparation complète entre les applications qu'elle embarque. La présence ou l'exécution d'un code malicieux n'a pas d'effet sur le système carte et ne représente donc pas une menace éventuelle pour les applications embarquées. Chaque application étant confinée dans son propre espace mémoire, elle ne pourra donc pas accéder aux données et aux codes des autres programmes installés. Ces différents mécanises sont évalués et certifiés par des organismes habilités (par exemple le DCSSI en France [40]), selon des normes et des critères de sécurité internationaux.

Quant à la confidentialité de l'application, les protections matérielles et logicielles intégrées dans la carte en font qu'une fois embarqué, le code de l'application est inaccessible au propriétaire du matériel qui possède pourtant un accès physique à la carte.

L'intégrité de l'exécution des applications embarquées sur la carte est assurée grâce à des mécanismes physique. En effet, les attaques qui ont pour but d'altérer ou de perturber l'exécution du programme sont généralement menées en appliquant des valeurs de tension ou de température en dehors des normes autorisées. La présence sur la carte de détecteurs embarqués permet de repérer toute utilisation dans des conditions anormales, et de bloquer l'application (voire la carte si nécessaire).

Finalement, pour ce qui concerne les communications, les protections matérielles et logicielles dont bénéficie la carte à puce permettent de fournir un espace de stockage hautement sécurisé pour les informations secrètes, espace dans lequel elles sont protégées contre les attaques d'accès non autorisée.

4.4 Une architecture générique pour les grilles de cartes à puce

L'infrastructure grille de cartes à puce peut être considérée comme une agrégation de clusters de cartes comme le montre la figure 4.1. Chaque cluster est géré par une machine pilote. Un utilisateur a accès aux cartes disponibles à travers le système grille.

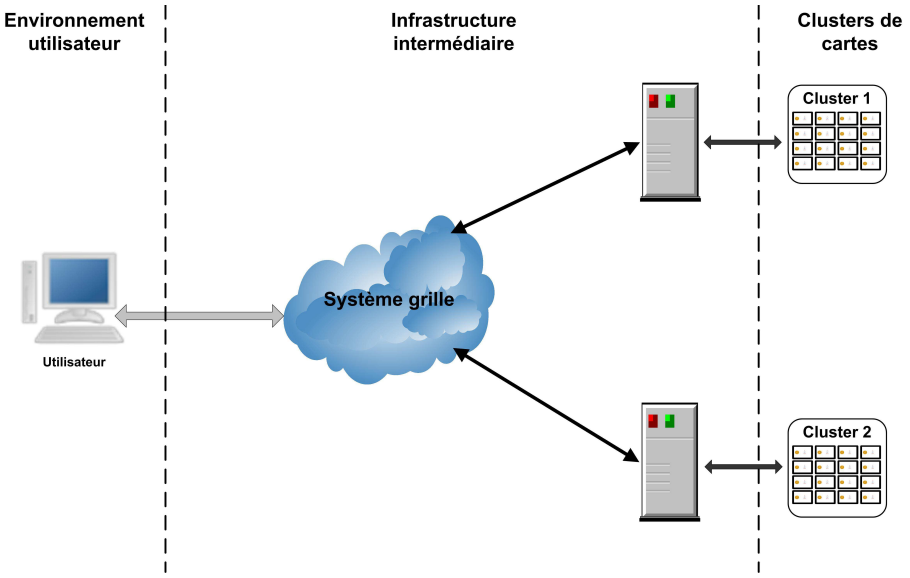


FIG. 4.1 – Vue globale de la grille de cartes à puce

4.4.1 Gestion des cartes à puce

Aujourd'hui, plusieurs entreprises (telles que les banques) ont recours à un système de gestion de parc de cartes, appelé CMS (*Card Management System*), afin de réaliser une administration globale de cartes déployées. Les fonctionnalités d'un CSM peuvent se résumer à :

- La gestion des commandes de cartes : il s'agit de commander les cartes auprès de fournisseurs et de choisir le type de carte le plus approprié selon le type de l'application (par exemple format des cartes, nature du support plastique, etc.).
- La personnalisation des cartes : il s'agit d'installer sur la carte les données concernant le porteur (l'utilisateur final) de la carte. Ces informations peuvent être par exemple un code secret, une clé ou un certificat d'authentification, des données personnelles (telles que le nom de l'utilisateur, sa date de naissance,...), etc. Il est également possible de personnaliser le support plastique de la carte (impression du logo de l'entreprise, de la photo de l'utilisateur, etc.).
- La gestion de la fin de vie de la carte : il s'agit d'inhiber les informations secrètes contenues dans la carte. Par exemple, le certificat de l'utilisateur chargé dans la carte doit être révoqué. La carte peut atteindre la fin de son cycle de vie par invalidation logique, vol, perte, etc.

4.4.2 Un cluster de cartes à puce

Ultérieurement à cette thèse, le LaBRI a mis en place une plate-forme de cluster de cartes à puce qui consiste à assembler (à la manière d'un cluster) un certain nombre de cartes à puce. Initialement, la plate-forme était constituée d'un seul PC permettant de gérer 9 cartes. L'ensemble des lecteurs était

"éparpillé" sur un bureau, comme le présente la photo 4.2. L'infrastructure a ensuite évolué jusqu'à comporter deux machines. La plate-forme est intégrée dans une armoire, présentée sur la photo 4.3, dans laquelle est déployé l'ensemble des cartes. L'armoire est composée de 19 unités, comportant :

- un PC occupant deux unités,
- deux unités *SmartMount*. Sur chacune d'elles sont placés 8 lecteurs de cartes CCID [25], de marque *SCM Microsystems*,
- 3 hubs USB placés à l'intérieur de l'armoire, et occupant deux unités. Ces hubs servent à connecter les lecteurs aux PC ce qui permet de disposer de 16 lecteurs par machine.

Cette configuration est reproduite en deux exemplaires et les deux serveurs sont reliés entre eux à travers un réseau local.

Fig. 4.2 – La plate-forme initiale

Fig. 4.3 – La plate-forme finale

L'armoire est de plus équipée d'un moniteur, ainsi que d'un clavier et d'un commutateur KVM (*Keyboard-Video-Mouse*). Nous disposons également d'une cinquantaine de cartes provenant de plusieurs fabricants.

4.4.3 Définition d'une architecture pour une grille de cartes à puce

Pour pouvoir expliquer par la suite les concepts fondamentaux pour la mise en place d'une architecture logicielle pour une grille de cartes à puce, nous définissons une architecture en couches permettant de présenter les briques de base pour de telle grille, comme le montre la figure 4.4. Certaines fonctionnalités décrites par notre architecture sont également présentes sur les grilles classiques. Ce sont toujours les mêmes problèmes et challenges qui préoccupent les concepteurs, quelle que soit la nature de la grille.

L'infrastructure matérielle constitue ce que nous appelons la couche inférieure de cette architecture. Elle est en dehors du périmètre de cette thèse. Elle comprend des clusters de cartes, des équipements réseaux, etc. La couche intergiciel permet de fournir les mécanismes et les procédures pour gérer les ressources disponibles et établir des communications sécurisées avec l'ensemble des cartes présentes. La couche suivante englobe un certain nombre d'outils supplémentaires offrant des fonctionnalités et des services pour les applications qui forment la couche de niveau le plus haut.

La couche matérielle : la couche matérielle représente l'infrastructure physique de la grille. Cette couche comprend tout le matériel constituant la grille, ainsi que les réseaux de transmission et les dispositifs de communication. Elle peut être découpée en deux niveaux, comme présenté sur la figure 4.5 :

- la partie inférieure correspond à la couche connectique qui permet d'assurer l'interconnexion des ressources. Elle comporte les équipements réseaux tels que les cartes *Ethernet*, les câbles physiques, etc. Considérés comme des matériels de connexion avec la carte à puce, les lecteurs de cartes font également partie de cette couche.

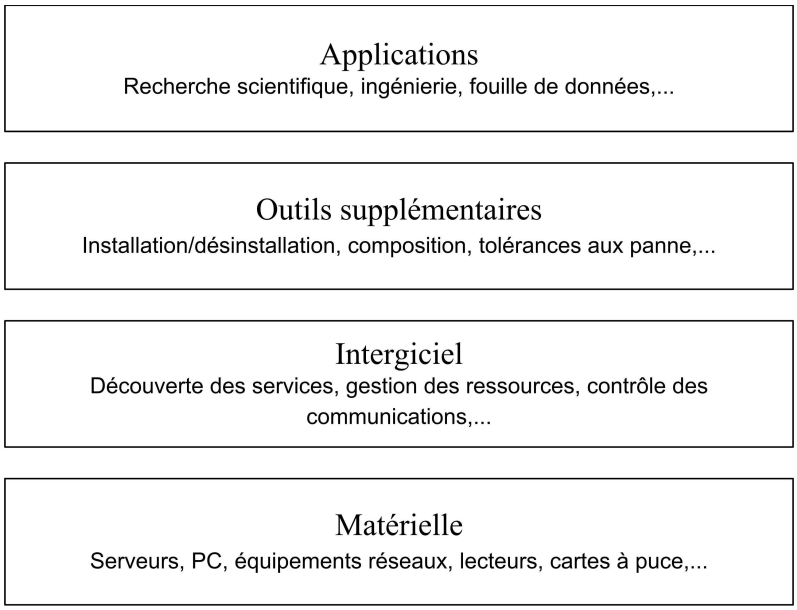


FIG. 4.4 – Architecture générale de la grille

- la couche machine comprend le serveur de données (qui héberge une sorte d'annuaire contenant des informations sur les cartes disponibles), les PC sur lesquels sont connectés les lecteurs, et bien sûr les cartes à puce, etc.

La couche intergiciel : la couche intergiciel assure les fonctions de gestion et de contrôle de la grille. Elle permet à un utilisateur de découvrir les ressources disponibles et d'y accéder facilement. En jouant le rôle de pont entre l'utilisateur et l'infrastructure matérielle, la couche intergiciel permet de rendre accessibles les ressources présentes en identifiant les cartes qui sont disponibles et en assurant les fonctions de communication de bas niveau. Elle fournit ainsi un accès uniforme à l'ensemble des cartes. Cette couche peut être découpée en deux parties, comme le montre la figure 4.5 :

- une couche inférieure composée de deux modules principaux. Le premier concerne la gestion de la topologie de la grille (découverte de l'apparition et de la disparition des clusters et des cartes), la gestion des ressources, etc. Le deuxième module fournit les mécanismes nécessaires à l'établissement de canaux sécurisés et à la protection des messages échangés.
- une couche supérieure qui permet d'assurer la gestion des communications et de maintenir et de fournir les informations concernant la topologie de la grille (informations fournies par la couche inférieure).

Les outils supplémentaires : à travers la mise en place de la couche outils supplémentaires, nous cherchons à répondre aux besoins les plus fondamentaux des utilisateurs de la grille. Nous proposons donc des solutions techniques pour quelques uns des problèmes que peuvent rencontrer le développeur d'applications ou l'utilisateur. Cette couche, qui est une couche optionnelle, regroupe un certain nombre d'utilitaires permettant d'apporter une assistance précieuse pour le déploiement et l'exécution des applications. La liste des utilitaires fournis pourra croître au fur et à mesure que de nouveaux besoins apparaîtront. Dans une première étape, nous avons identifié un certain nombre de fonctions que nous avons jugées essentielles.Ces outils sont : la composition des Card Services (voir section 4.5.1), un utilitaire graphique pour installer des applications, un ordonnanceur tolérant aux pannes pour gérer l'exécution d'une application parallèle. Le chapitre 7 sera consacré à la présentation de ces outils.

La couche applicative : au niveau le plus haut de notre architecture, figure la couche applicative qui est définie par l'utilisateur final.

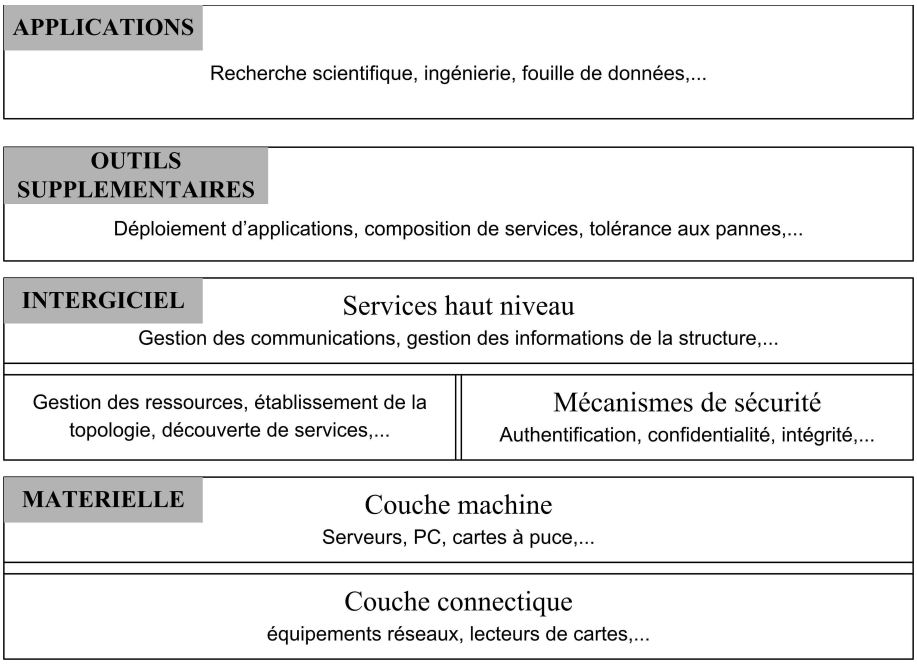


Fig. 4.5 – Architecture détaillée de la grille

Rappelons ici que notre ambition n'est pas de faire des calculs intensifs à travers l'utilisation d'une grille de cartes à puce, mais plutôt de se concentrer sur les aspects de sécurité fournis par la technologie carte à puce [30, 32]. Dans une grille classique, les ressources mises à disposition possèdent une capacité de calcul et de mémoire importante favorisant l'exécution de n'importe quelle application. Ceci n'est pas vrai pour une carte à puce. Les applications déployées doivent tenir compte des limitations de mémoire et de vitesse du processeur. De plus, les types supportés par les cartes sont très limités (la plupart de cartes ne supportent pas les nombres à virgules flottantes). En contre partie, la carte à puce bénéficie d'une sécurité logicielle et matérielle permettant de satisfaire les besoins des applications où l'aspect sécurité prime sur l'aspect calcul.

4.5 La couche intergiciel

4.5.1 La notion de Card Service

Afin de faciliter l'intégration de la carte à puce dans les systèmes distribués, nous avons introduit la notion de Card Service. Nous avons donc adopté deux approches différentes, mais complémentaires. La première approche permet de se focaliser sur les ressources matérielles proprement dites, c'est-à-dire les cartes. La seconde approche permet de considérer la grille comme étant une agrégation d'entités logicielles, appelées *Card Services*. La mise en œuvre de l'aspect service sera détaillée plus tard dans la section 4.6 ; nous nous limiterons ici à présenter ce qu'est un Card Service.

Nous avons défini un Card Service comme étant une application carte (une applet au sens Java Card) qui fournit un ensemble d'opérations ou de fonctionnalités. Le fournisseur d'un service doit écrire un fichier décrivant les opérations offertes, l'AID de l'application implémentant le service, le mode d'interaction avec ce service, etc. Cette description est enregistrée sur la carte auprès du catalogue embarqué qui n'est autre qu'une application carte dédiée comme le montre la figure 4.6. Cet annuaire contient les descriptions des Card Services qui sont installés. Un client pourra télécharger la description d'un service donné et utiliser les opérations qu'il fournit.

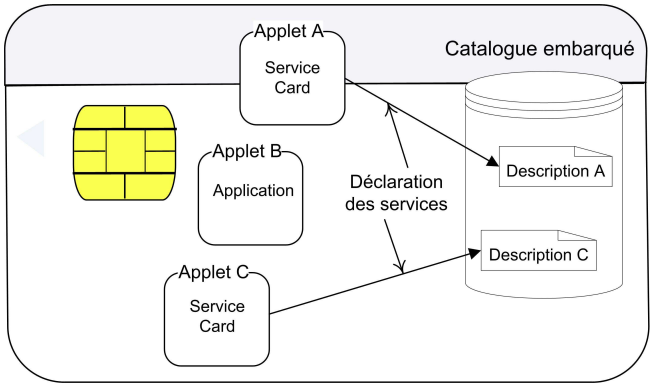


FIG. 4.6 – Définition d'un annuaire embarqué pour les Card Services

4.5.2 Les fonctionnalités offertes par l'intergiciel

La couche intergiciel assure plusieurs fonctionnalités de base liées essentiellement à la gestion des ressources, des Card Services et des communications. Elle permet de construire et de gérer la structure globale du système afin de rendre les cartes formant la grille accessibles aux utilisateurs.

4.5.2.1 Le système d'information

Dans un environnement de type grille, les ressources disponibles sont réparties à travers un ou plusieurs réseaux et leur nombre peut varier au cours du temps. Un des objectifs de la conception de la grille de cartes à puce est de rendre les ressources (en l'occurrence les cartes) facilement accessibles aux utilisateurs. Il est donc nécessaire de fournir les moyens appropriés permettant d'enregistrer et de découvrir les cartes disponibles, et de déterminer leur état. La couche intergiciel, à travers une entité de publication, permet de gérer des informations sur la structure globale du système. Les informations stockées au niveau de l'entité de publication peuvent être de nature statique ou dynamique. L'entité de publication contient par exemple des données relatives à la localisation et au contenu des cartes (liste des AID et fichiers de description des Card Services). D'autres informations complémentaires permettant de décrire les cartes peuvent aussi être stockées : propriétés physiques, caractéristiques matérielles, modèle, capacité mémoire et de calcul, etc. Il est également possible de stocker des informations relatives à la disponibilité des cartes (libre, en cours d'utilisation, hors service, etc.). Pour cela, il est essentiel de définir un système d'identification et un système de localisation des différentes cartes. Nous décrivons maintenant ces aspects.

Identification des cartes : comme tout système distribué, une grille doit pouvoir référencer ses ressources de façon uniforme. Nous définissons le système d'identification suivant. À chaque carte est attribué un pseudonyme (nom alphanumérique) permettant de l'identifier de façon unique dans la grille. Le nom de la carte peut être enregistré dans des fichiers spéciaux dans lesquels figurent d'autres informations (ses propriétés physiques, ses capacités matérielles, l'identifiant de son certificat, etc.). Ces fichiers sont stockés sur la carte. Une autre solution (que nous avons écartée) consisterait à considérer le CSN (*Card Serial Number*) comme identifiant. Plusieurs recommandations déconseillent l'utilisation de ce numéro de série à des fins d'identification [146, 39]. D'après les définitions présentées par le consortium *Smart Card Alliance* [132], consortium regroupant des acteurs industriels du domaine des cartes à puce, l'unicité du numéro de série est en fait lié à un ensemble donné de cartes. Il est donc possible (même si la probabilité est très faible) d'avoir deux cartes qui ont le même CSN[4]. Afin d'éviter toute incohérence, il est donc vivement conseillé d'attribuer manuellement un pseudonyme qui sera par la suite chargé

[4]Dans la majorité des cas, le CSN est fixé durant la phase de fabrication de la carte et ne peut pas être modifié par la suite. Cependant, pour certains modèles, il est possible de changer le CSN de façon logicielle après la mise en circulation de la carte (par exemple par l'utilisateur final).

sur la carte. Chaque carte aura alors son propre identifiant unique que nous notons CID (*Card IDentifier*).

Localisation des cartes : pour pouvoir communiquer avec une carte, il faut tout d'abord déterminer son emplacement physique. Or, la carte à puce dépend d'un lecteur appelé *Card Accepting Device* pour être opérationnelle. Ce dernier doit être relié à une machine donnée dans le réseau. La localisation d'une carte se fait donc à travers la machine et le lecteur dans lequel elle est insérée. Le triplet <adresse de machine, nom de lecteur, identifiant de carte> fournit les informations complètes sur la localisation de la carte.

4.5.2.2 Gestion des ressources

Comme toute infrastructure de type grille, la grille de cartes à puce est une architecture dynamique : les ressources (en l'occurrence les cartes) peuvent apparaître et disparaître à n'importe quel moment. Pour pouvoir déterminer l'ensemble des cartes disponibles, il est nécessaire de mettre en œuvre un gestionnaire de ressources qui permette de découvrir dynamiquement les cartes qui sont présentes ainsi que leur état de disponibilité.

La fonctionnalité principale du gestionnaire de ressources est de contrôler la structure globale de la grille et de détecter toute évolution de sa topologie. À cette fin, chaque machine responsable d'un cluster doit être capable de gérer les cartes qui y sont connectées. Tout événement lié à une carte (insertion, arrachage) ou à un lecteur (débranchement) doit être intercepté. Le gestionnaire de ressources doit également tenir compte d'éventuels événements pouvant se produire au niveau des machines pilotant les clusters. Des mécanismes sont donc mis en place permettant de détecter, de gérer et de traiter l'arrêt/démarrage des machines hébergeant les clusters de cartes. L'entité de publication doit être informée chaque fois qu'un événement se produit afin de maintenir des informations conformes à la structure globale de la grille.

4.5.2.3 Construction des Card Services

D'une façon générale, les architectures Service Web [20] sont basées sur trois acteurs (ou rôles) principaux, à savoir, le fournisseur de service, le client d'un service et l'annuaire de services. Trois actions principales sont également définies : découvrir, publier et invoquer.

Un fournisseur qui propose un service doit en fournir une description. Cette dernière est publiée à travers l'annuaire des services ce qui permet aux clients de consulter la description des services disponibles. En effectuant une recherche, le client peut découvrir le ou les services répondant à ses besoins. Une fois le fichier de description récupéré, le client est capable d'accéder au service et d'invoquer ses opérations. La figure 4.7 présente le fonctionnement global des Web Services.

Le fait d'appliquer l'aspect service pour les cartes à puce soulève des problèmes au niveau de la publication et de la description des services. En effet, les supports d'exécution des services, en l'occurrence les cartes, possèdent un caractère dynamique vu leur portabilité : une carte peut apparaître et disparaître à tout moment. Cette dynamicité engendre des problèmes lors de la définition de la description d'un Card Service. L'étude de ce problème et les solutions proposées seront détaillées dans la section 4.6.

4.5.2.4 Système de communication

Dans une architecture distribuée, les communications jouent un rôle crucial. Dans notre plateforme, nous cherchons à ce que l'utilisateur puisse accéder à distance aux différentes cartes disponibles. Or, la communication avec une carte à puce obéit au protocole APDU. Ce protocole définit le mode de communication entre la carte et le terminal auquel elle est directement liée. Par conséquent, la carte à puce ne supporte pas les techniques de communication distante de type RMI[5], Corba, etc. Afin

[5] À partir de la version 2.2, les cartes Java intègrent JCRMI (*Java Card Remote Method Invocation*) permettant d'appeler à distance les méthodes d'une application embarquée. Cette fonctionnalité est mise en place à travers un proxy du côté terminal qui traduit les invocations de méthodes en leurs APDU correspondantes. Néanmoins, ceci étant propre aux cartes Java et ne fait pas l'objet d'un standard commun pour tous les autres types de cartes.

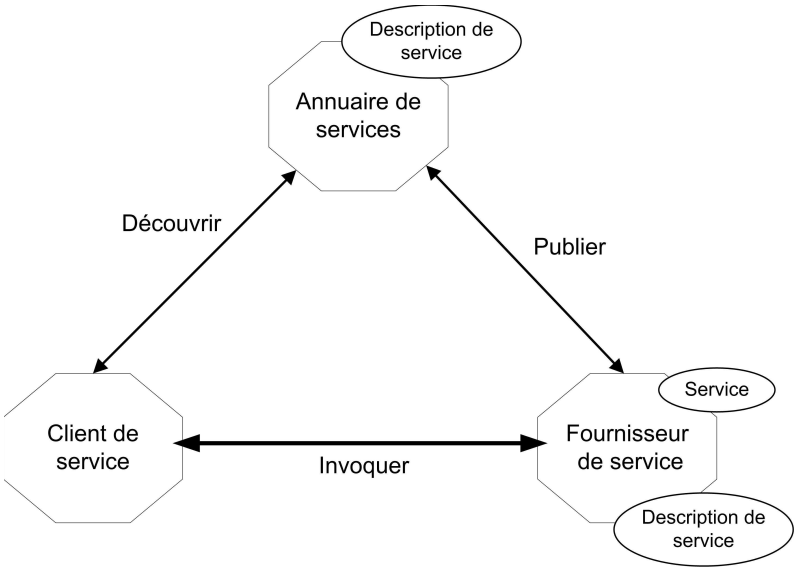


FIG. 4.7 – Fonctionnement des Web Services

de remédier à ce problème, la communication entre l'utilisateur et une carte donnée s'établit en deux étapes. Dans une première phase, un canal de communication doit être créé entre le client et le terminal où se trouve le lecteur de carte. Les différentes technologies des systèmes distribués (telles que RMI, Corba, les Web Services, etc.) permettent d'assurer cette première phase. Dans une seconde étape, le terminal doit établir une communication directe avec la carte concernée. Cette communication utilise le protocole APDU.

La proactivité de la carte

Un autre aspect que doit également traiter le système de communication est celui de la proactivité. De par sa conception, la carte à puce est un périphérique passif : elle n'est pas capable de prendre l'initiative d'appeler un code à l'extérieur de son environnement d'exécution. Dans ce schéma de communication, dit client-serveur, la carte est toujours le serveur qui attend les requêtes des clients (commandes APDU), pour exécuter les opérations correspondantes et retourner une réponse APDU. Les applications utilisant notre infrastructure pourraient avoir besoin d'un modèle dans lequel la carte soit pro-active, c'est-à-dire qu'elle soit capable d'appeler une application extérieure. Dans les cartes SIM des téléphones portables (appelées aussi USIM) par exemple, ce problème a été résolu grâce à l'interrogation périodique de la carte par le terminal [100]. Selon les valeurs des champs d'état reçues dans la réponse de la carte, le terminal peut savoir si celle-ci veut envoyer une commande pro-active. Si c'est le cas, il lui renvoie une commande dite *Fetch Command*, ce qui permet à la carte d'envoyer sa requête pro-active comme illustré à la figure 4.8.

De façon similaire, nous proposons dans le chapitre 5 une solution basée sur les champs d'état permettant à la carte d'engager une communication et d'envoyer des "commandes pro-actives". Dans ce modèle d'exécution, la carte ne serait plus seulement serveur, mais également client. Nous définissons deux formes de proactivité, une proactivité forte permettant à une carte d'appeler un code sur une autre carte, et un mode de proactivité faible qui permet à la carte d'exécuter un code sur une machine classique. Plus de détails seront donnés dans la section 5.5.

Sécurité des communications

Comme nous l'avons expliqué dans les chapitres précédents, les cartes à puce possèdent des protections physiques et logicielles permettant d'assurer la confidentialité du code et l'intégrité de l'exécution des applications qui y sont installées. De plus, les cartes à puce bénéficient d'une architecture sécurisée

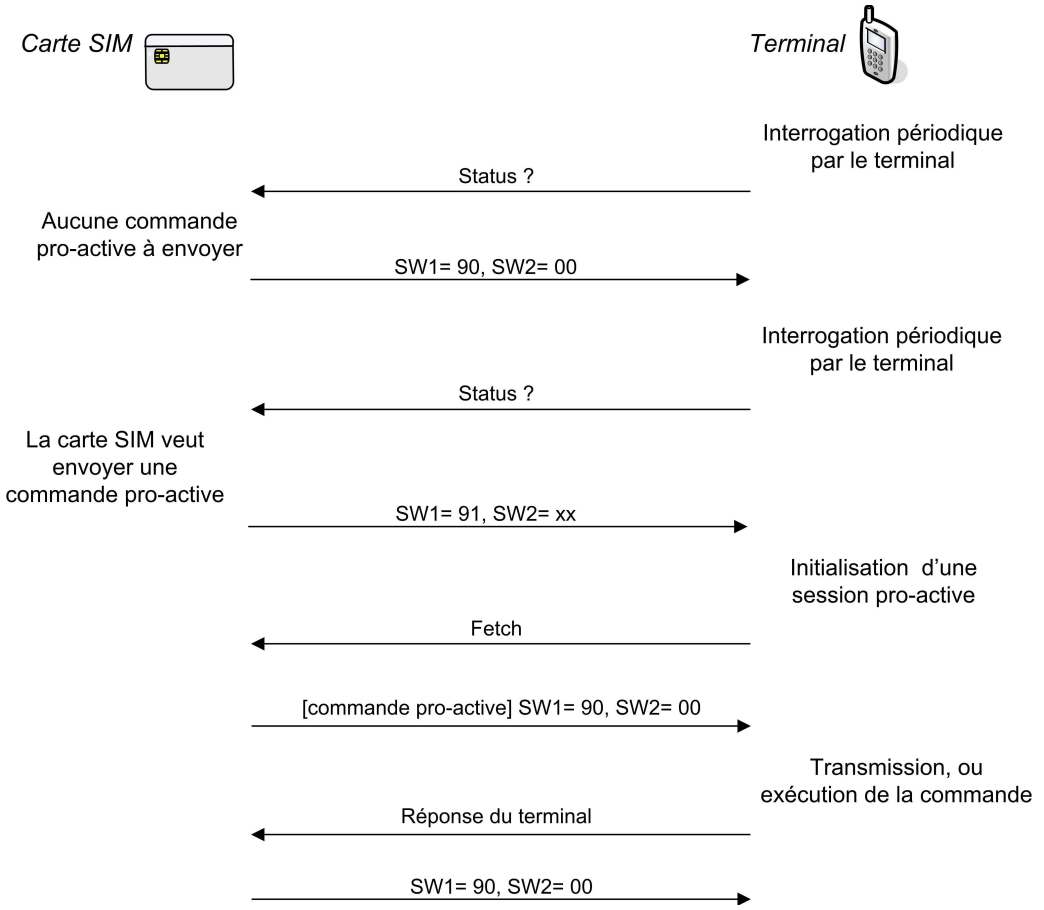


Fig. 4.8 – Envoi d'une commande pro-active pour les cartes SIM (d'après [100])

(notamment une API pour les fonctions cryptographiques) ; de ce fait la sécurité des communications inter-cartes peut en découler directement. Il ne reste donc plus que la sécurité des communications à garantir.

Afin de protéger les informations échangées contre d'éventuelles attaques, une politique de sécurité doit être mise en place permettant de définir un protocole de communication sécurisé entre l'utilisateur et la carte. Cette protection consiste à assurer la confidentialité des données échangées, l'intégrité des messages transmis et l'authentification des interlocuteurs. Le protocole que nous proposons permet d'établir des canaux sécurisés entre un client et une carte à la manière de GlobalPlatform. Dans le chapitre 6 nous présentons en détails le protocole de communication que nous avons conçu.

4.6 Intégration de la carte à puce dans les systèmes distribué

Pour permettre une meilleure intégration de la carte à puce dans les systèmes distribués, plusieurs solutions ont été proposées. Ces solutions offrent une abstraction vis à vis du protocole de communication APDU qui constitue le vrai handicap pour l'utilisation d'une carte comme un nœud autonome dans un environnement distribué. En effet, pour pouvoir communiquer avec une application carte, un utilisateur est obligé de passer par les messages APDU. Afin d'éviter cette contrainte, un proxy est mis en place entre l'application carte et l'application cliente permettant de convertir les requêtes appropriées en messages APDU et de les envoyer vers la carte. Des solutions telles que JiniCard [95] (basée sur la technologie JINI), ORBCard [27] pour la technologie Corba ou JCRMI [133] ont déjà été développées pour communiquer de manière transparente avec les applications embarquées sur la carte

à puce. Récemment, le standard Java Card introduit dans sa dernière version 3.0 [89] une nouvelle machine virtuelle Java Card qui supporte des aspects orientés réseaux et les applications Web. Les spécifications décrivent l'utilisation d'un serveur Web embarqué dans la carte qui peut être accessible depuis une application extérieure. Il s'agit plus précisément d'une API encartée pour le développement de servlets à l'intérieur de la carte.

4.6.1 Les travaux existants

Nous avons identifié dans la littérature un certain nombre de projets visant à intégrer des cartes Java dans un environnement distribué. Nous les présentons ci-dessous.

ORBCard [27] est une plate-forme dont l'objectif est d'intégrer la carte à puce dans les systèmes répartis en se basant sur la technologie distribuée CORBA [77]. Cette approche est mise en œuvre à travers une couche logicielle qui permet d'interagir avec les applications présentes sur la carte, comme si elles étaient des objets Corba classiques. Cette architecture est basée sur le composant *OrbCard adaptor* qui joue le rôle de passerelle entre la carte (le serveur) et son milieu extérieur (le client) comme illustré sur la figure 4.9.

FIG. 4.9 – L'approche ORBCard : utilisation de la technologie CORBA pour les cartes à puce

JiniCard [95] utilise l'environnement Jini [42] pour permettre aux applications de découvrir et d'invoquer les services proposée par une carte. Comme Jini, JiniCard est basé sur un serveur catalogue (*lookup*) qui contient la liste des applications encartées. L'interaction avec le service encarté est réalisée à travers une application intermédiaire, installée sur le terminal auquel la carte est connectée, et qui permet d'invoquer les méthodes proposées par la service encarté comme le montre la figure 4.10.

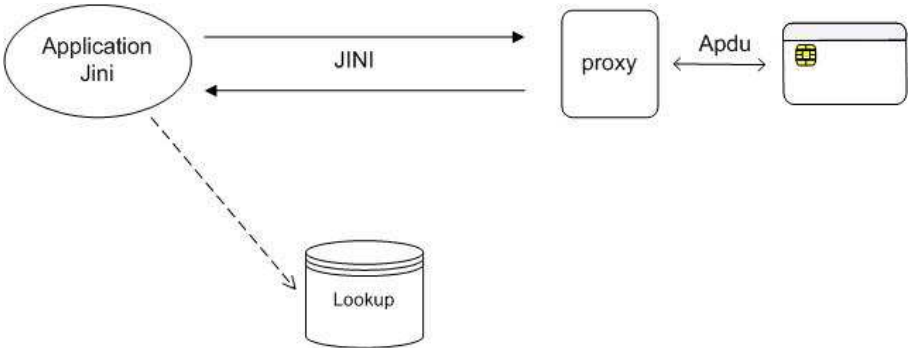


FIG. 4.10 – L'approche JiniCard

JCRMI [133] est apparu avec la version 2.2 de la norme Java Card. Le but de ce standard est de proposer la technologie RMI, largement utilisée dans le cadre des systèmes distribués, pour permettre d'invoquer à distance les méthodes d'une application encartée comme présenté à la figure 4.11.

La carte hybride [92]. Dans cette approche en plus de sa capacité d'exécution des applications, la carte est considérée comme une base de données capable d'exécuter des requêtes SQL (récupération, suppression ou mise à jour des données). Elle intègre un moteur embarqué pour

Fig. 4.11 – L'approche Java Card RMI

l'exécution des requêtes SQL. Le concept de la carte hybride permet de supporter trois modes de fonctionnement comme présenté sur la figure 4.12 :

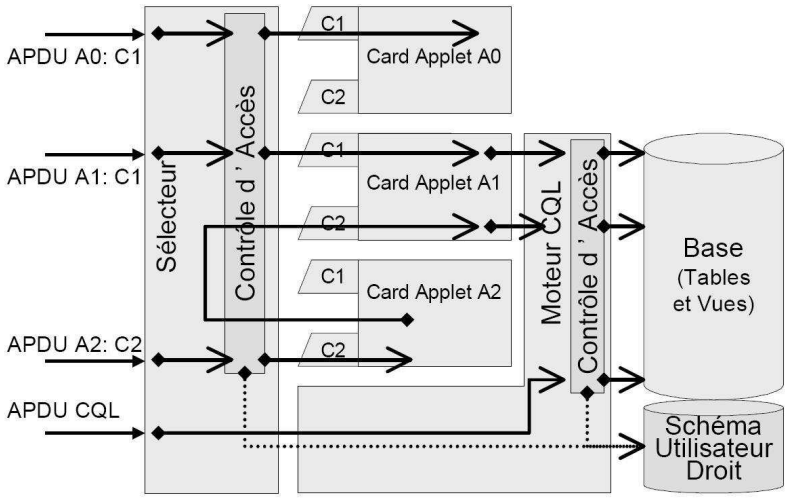


Fig. 4.12 – La carte hybride (source : [92])

- Carte base de données : la carte est utilisée en tant que carte base de données en envoyant des APDU CQL qui sont traitées par le moteur CQL interne.
- Carte multi-services : les APDU envoyées correspondent à des appels de service (cas de l'APDU A0 : C1). L'exécution de la commande est réalisée en utilisant uniquement des données persistantes propres à l'application se trouvant dans son contexte (espace mémoire attribué par le support d'exécution). Il s'agit en fait d'une utilisation classique d'une carte multi-applicative.
- Carte multi-services avec données structurées : les APDU envoyées correspondent toujours à des appels de service (cas des APDU A1 : C1 et A2 : C2), mais ces services sont conçus pour utiliser des données au travers de la base de données.

La WebCard [121]. Dans cette approche, la carte est considérée comme un serveur Web qui peut être accessible à travers des messages HTTP ce qui permet de la rapprocher au plus des technologies Web. Les requêtes HTTP sont envoyées vers la carte à travers un proxy directement lié à la carte. Le proxy agit comme intermédiaire entre la carte et le client et permet d'encapsuler les requêtes HTTP dans le champs données des commandes APDU comme présentée à la figure 4.13.

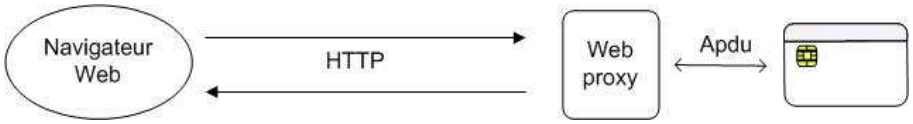


Fig. 4.13 – La carte comme serveur Web

Les servlets. Dans la version 3.0 de Java Card [89], sortie en mars 2008, Sun introduit une édition connectée de la plateforme Java Card. Cette version se caractérise par une nouvelle machine Virtuelle qui supporte des aspects orientés réseau (pile TCP/IP) et les applications Web (une

API encartée pour le développement de Servlets). Ces spécifications décrivent en particulier l'utilisation d'un serveur web embarqué dans la carte accessible depuis une application extérieure comme le montre la figure 4.14. Le grand avantage des servlets Java Card, est que la carte dispose d'une pile TCP/IP intégrée : elle possède donc une adresse IP et peut être vue comme un nœud dans le réseau.

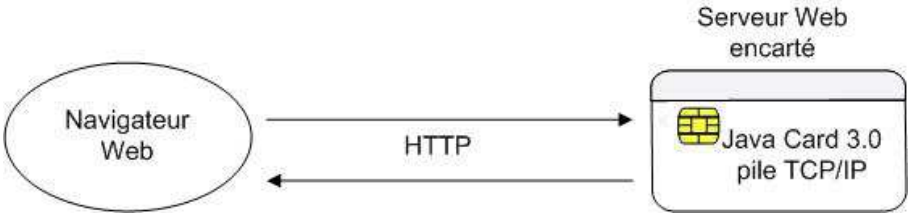


FIG. 4.14 – Les servlets dans la Java Card 3

Dans la suite de cette section, nous présentons l'approche que nous avons conçue et qui consiste à considérer l'application carte comme un *Card Service* à l'image des *Web Services* [20].

4.6.2 Un modèle basé services pour la carte à puce

Une architecture orientée service peut être définie comme une architecture d'interaction entre différents entités (notamment un fournisseur et un client), mettant en œuvre un composant logiciel, le service. Selon une définition empruntée à [16], *un service est un comportement défini par contrat, qui peut être réalisé et fourni par tout composant pour être utilisé par tout composant, sur la base unique du contrat.* Dans les architectures orientées service, le contrat d'un service comporte plusieurs thèmes ou parties. Nous distinguons principalement la description des fonctions du service, la description de l'interface du service, la description des termes de l'échange, etc. Dans la suite de cette section nous présentons notre modèle de *Service Card* en sa basant sur la technologie de Service Web qui est un représentant des architectures orientées service.

4.6.2.1 Définition des rôles

Nous présentons ici les différentes entités, représentées à la figure 4.17, qui interviennent pour la mise en œuvre d'une architecture Card Service :

- le fournisseur de service (*Service Provider*) : c'est l'entité (personne ou organisation) propriétaire du service. Elle réalise une implémentation du service et l'installe sur la carte. Le fournisseur de service doit également proposer une description technique permettant de connaître le mode d'interaction avec le service (opérations offertes, structure et format des requêtes, etc.).
- le client de service (*Service Requester*) : il s'agit d'un client potentiel qui a besoin des fonctionnalités offertes par le service. Cette entité peut être un utilisateur, une application ou même un autre service.
- l'annuaire global des services (*Global Service Registry*) : c'est une entité de stockage responsable de l'hébergement de la description des services disponibles et de leur publication. L'annuaire des services joue un rôle essentiel dans notre modèle puisqu'il fournit au client toutes les informations nécessaires pour accéder à un service donné.
- le service : il s'agit d'une entité logicielle (application) réalisant un ensemble d'opérations qui fournissent des fonctionnalités bien déterminées. L'application implémentant le service doit être chargée sur la carte. Il est aussi possible qu'un même service soit présent sur plusieurs cartes.
- la description de service : la description d'un service contient les informations permettant de communiquer avec le service : opérations offertes, structure des messages, données de localisation de la carte hébergeant ce service, etc. La description doit être publiée dans l'annuaire des services. Nous définissons deux types de descriptions : une description fonctionnelle et une description complète comme nous allons le voir dans la section 4.6.2.2.

– le catalogue embarqué ou encarté (*Embedded Registry*) : c'est une application annuaire encarté qui doit être chargée au préalable sur la carte. Elle maintient les descriptions des Card Services installés sur la carte.
– l'agent de supervision des cartes (*Monitor Agent*) : c'est un composant logiciel qui se trouve sur le terminal auquel la carte est directement liée. Sa fonction est de détecter l'insertion/arrachage des cartes et de récupérer la description des Card Services depuis l'annuaire encarté.

4.6.2.2 Description d'un service

Un *Card Service* est défini selon un schéma XML. Cette description comprend l'interface du service lui même (niveau fonctionnel) et la localisation des différentes cartes qui l'hébergent comme le montre la figure 4.15. À travers cette description, un service pourra être publié et invoqué.

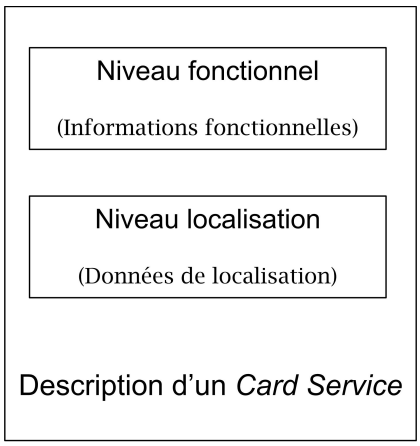


Fig. 4.15 – Format général du descripteur d'un *Card Service*

Au niveau fonctionnel, on décrit les détails techniques, à savoir l'AID de l'applet implémentant le service, les opérations ou les méthodes fournies, la valeur du champ instruction de la commande APDU correspondant à chaque opération, le format des messages, les paramètres d'entrée/sortie, etc. Le niveau localisation permet d'indiquer les cartes sur lesquelles le service est installé (adresse de la machine, nom du lecteur, identifiant de la carte). À partir de ces deux niveaux, nous définissons deux formats de fichiers distincts :
– une description minimale (ou fonctionnelle) : correspond à une description technique contenant uniquement la partie fonctionnelle. Cette description permet aux fournisseurs de définir le mode de fonctionnement de leurs services ;
– une description complète : récupérée par les clients potentiels de service, elle est composée d'une partie fonctionnelle et d'une partie localisation.

Le fournisseur de service doit spécifier la partie fonctionnelle de son service qui sera chargée par la suite sur la carte, dans l'annuaire embarqué. La partie localisation sera construite suivant la disponibilité du service. Ainsi, deux utilisateurs pourront recevoir deux descriptions différentes pour un même service : la partie fonctionnelle sera la même ; la différence concernera éventuellement la partie localisation qui dépend des disponibilités du service sur les différentes cartes.

4.6.2.3 Étapes de gestion des Card Services

Le modèle d'une architecture Card Service est basé sur trois actions fondamentales : publier, découvrir et invoquer. Ce sont les mêmes étapes que pour les architectures Services Web [20].

1. Publication des services

La publication est la fonction qui permet d'annoncer aux clients potentiels les services qui sont disponibles. Cette fonction est réalisée par l'application annuaire qui doit être accessible par tous les clients et qui contient les descriptions des services déployés.

Pour ce qui concerne les *Card Services*, la publication est réalisée en deux phases, une publication interne à la carte et une publication externe.

Publication interne : durant cette première étape, un fournisseur doit (1) installer l'application implémentant le service et (2) enregistrer son service auprès du catalogue local de la carte. L'application est désormais vue comme un Card Service. Il est essentiel de s'assurer de la réussite de l'installation de l'application avant de passer à l'étape d'enregistrement. De plus, si l'étape d'enregistrement échoue le service ne pourrait pas être publié.

Publication externe : la publication externe a pour objectif de rendre la description des services disponibles accessible aux clients. Deux étapes sont nécessaires :

- récupération des services : cette étape consiste à récupérer la description des services installés sur les cartes. Ceci est réalisé par l'agent de supervision qui interroge l'annuaire embarqué sur la carte afin d'en extraire la liste des descriptions qui ont été chargées lors de la phase de publication interne. Rappelons qu'une application ne peut être considérée comme un service que si elle a été déclarée comme tel au moment de son installation sur la carte, c'est-à-dire qu'elle a été enregistrée auprès du catalogue embarqué. Les informations récupérées sont ensuite envoyées à l'annuaire global des services.
- enregistrement des services : lors de cette étape, les descriptions techniques (ou minimales) des *Card Services* disponibles qui ont été téléchargées depuis les cartes sont envoyées vers l'annuaire global où elles sont sauvegardées. En plus de la description minimale, l'agent de supervision doit aussi préciser dans sa requête d'enregistrement les données relatives à la localisation du service, ce qui permet à l'annuaire global de construire une description complète.

Notons que si un service disparaît suite par exemple à l'arrachage de la carte qui le propose, il faut mettre à jour les informations dans l'annuaire global. Cette fonction est réalisée par l'agent de supervision qui est responsable de la détection de l'insertion/arrachage des cartes, et par conséquent la détection d'apparition/disparition des services. Il doit donc informer l'annuaire global des événements survenus pour mettre à jour les informations des Card Services.

2. Découverte des services par les utilisateurs

Pour découvrir les services disponibles répondant à ses besoins, un client envoie à l'annuaire global une requête (de découverte) afin de chercher un service réalisant une fonctionnalité bien déterminée. L'annuaire renvoie alors la description complète (comportant la partie fonctionnelle et la partie localisation) du ou des Card Services répondant à la requête reçue. En se basant sur les informations contenues dans le fichier de description, le client peut alors accéder au service en question.

3. Invocation de services

Pour communiquer avec un service donné et invoquer les opérations qu'il fournit, un client doit se baser sur les informations contenues dans le fichier de description récupéré lors de la phase de découverte. Pour simplifier la tâche, nous avons mis en œuvre un outil graphique permettant à l'utilisateur de découvrir la liste des Card Services disponibles et d'invoquer un service choisi. L'utilisateur n'a qu'à préciser les paramètres d'entrée et l'invocation du service est réalisée de manière automatique comme le montre la figure 4.16. Nous présentons également dans le chapitre 7 un outil de composition de services, tolérant aux fautes, permettant d'invoquer plusieurs Card Services en une seule requête.

La figure 4.17 présente les différentes étapes, que nous venons de décrire pour la mise en œuvre d'une architecture Card Service.

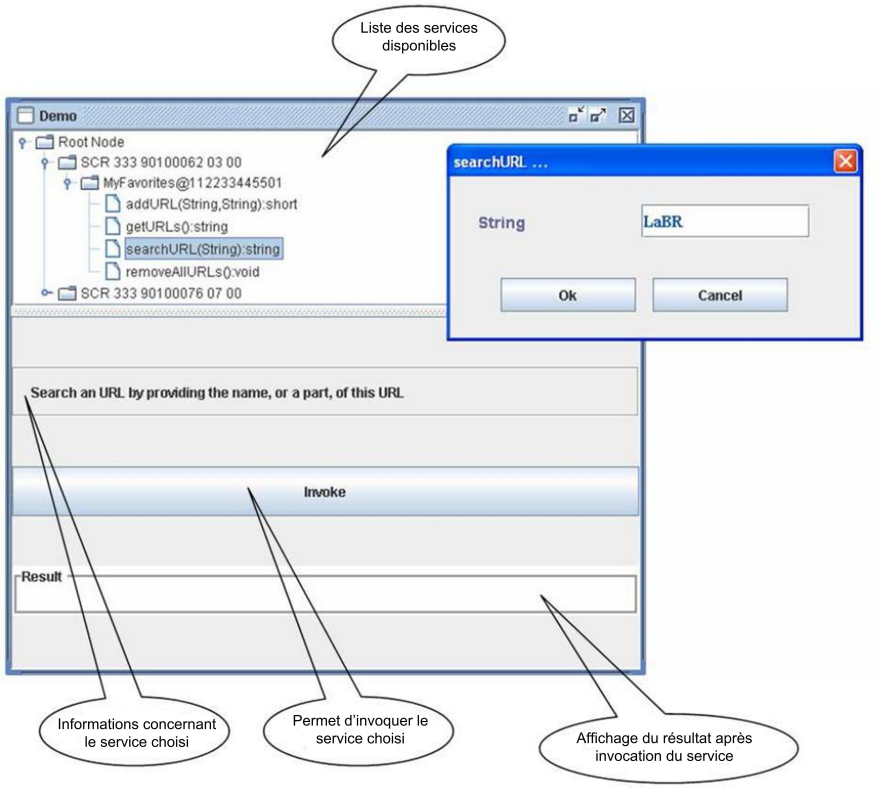


Fig. 4.16 – Outil pour l'invocation des Card Services

4.7 Conclusion

Dans ce chapitre, nous avons établi une architecture de référence pour les grilles de cartes à puce. Cette architecture, définie en terme de couches, permet de présenter des concepts fondamentaux pour les plate-formes de type grilles de cartes à puce. Un certain nombre de problèmes, qui ne se posent pas dans les grilles classiques, ont été identifiés et décrits. Ces problèmes concernent principalement la localisation et la proactivité des cartes. Nous avons également introduit la notion de Card Service permettant une meilleure intégration des cartes à puce dans les systèmes distribués. La grille de cartes à puce peut alors être considérée comme un ensemble de ressources matérielles (les cartes) ou logicielles (Card Services). Les solutions techniques apportées aux problèmes évoqués dans ce chapitre seront présentées dans la suite de ce document.

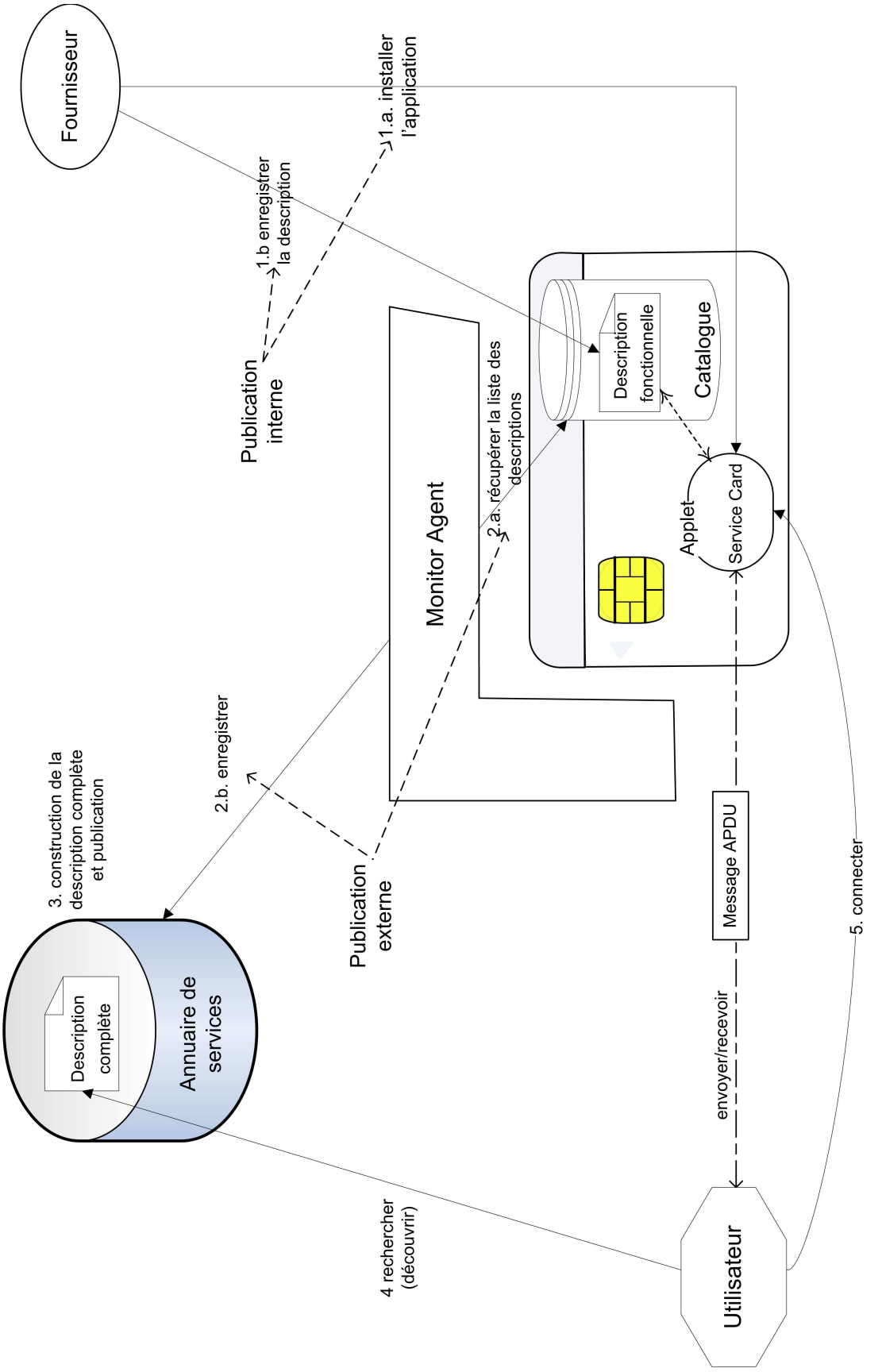


FIG. 4.17 – Mise en œuvre des Card Services

Chapitre 5

La couche intergiciel

Sommaire

5.1 Introduction

La mise en œuvre d'un environnement de type grille repose principalement sur la conception de la couche intergiciel qui est considérée comme le cœur du système. Cette partie est fondamentale car elle constitue un pont entre l'architecture matérielle et l'utilisateur. Dans ce chapitre nous présentons un modèle d'implémentation pour la couche intergiciel dans le contexte d'une grille de cartes à puce de type Java en décrivant les principales fonctionnalités fournies.

5.2 Une grille de cartes à puce de type Java : vue d'ensemble

Le modèle d'une plate-forme de cartes à puce de type Java est basé sur un certain nombre de composants qui interagissent entre eux selon des schémas de communication bien déterminés. Nous détaillons dans cette section ces différents constituants et nous définissons le rôle de chacun d'entre eux.

5.2.1 Les composants de la plate-forme Java Card Grid

La grille de cartes à puce Java est mise en place à travers un ensemble d'entités, représentées sur la figure 5.1. Nous distinguons :

– un ou plusieurs serveurs cartes (Cluster-Manager) : c'est le serveur ou la machine qui pilote un cluster de cartes. Chaque cluster est formé d'un certain nombre de cartes qui sont reliées par des hubs USB. Le Cluster-Manager établit des voies de communication directes avec toutes les cartes.
– registre de configuration : il constitue le système de données pour la grille et permet d'assurer une fonction de publication. Les données stockées concernent la structure globale de la grille, en particulier des cartes. Ces données sont mises à jour chaque fois que la structure ou l'état de la grille change. Le registre de configuration fournit les moyens pour accéder de manière uniforme aux informations qu'il contient. Ces données peuvent être répertoriées en deux catégories :
    • des données statiques : le nom des cartes, leur type et leur modèle, leur CSN (*Card Serial Number*), leurs caractéristiques matérielles (taille des mémoires, type du processeur), leur certificat, etc.
    • des données dynamiques : le contenu des cartes (c'est-à-dire la liste des applications et des Card Services qui y sont installées), des données de localisation (formées de triplets <machine, lecteur, nom>), des informations de disponibilité, etc.
– un serveur proxy : constitue une passerelle entre les utilisateurs et les cartes. Lorsqu'un client souhaite accéder à une carte donnée, il doit passer par le proxy. Il s'agit d'un élément central de la grille.

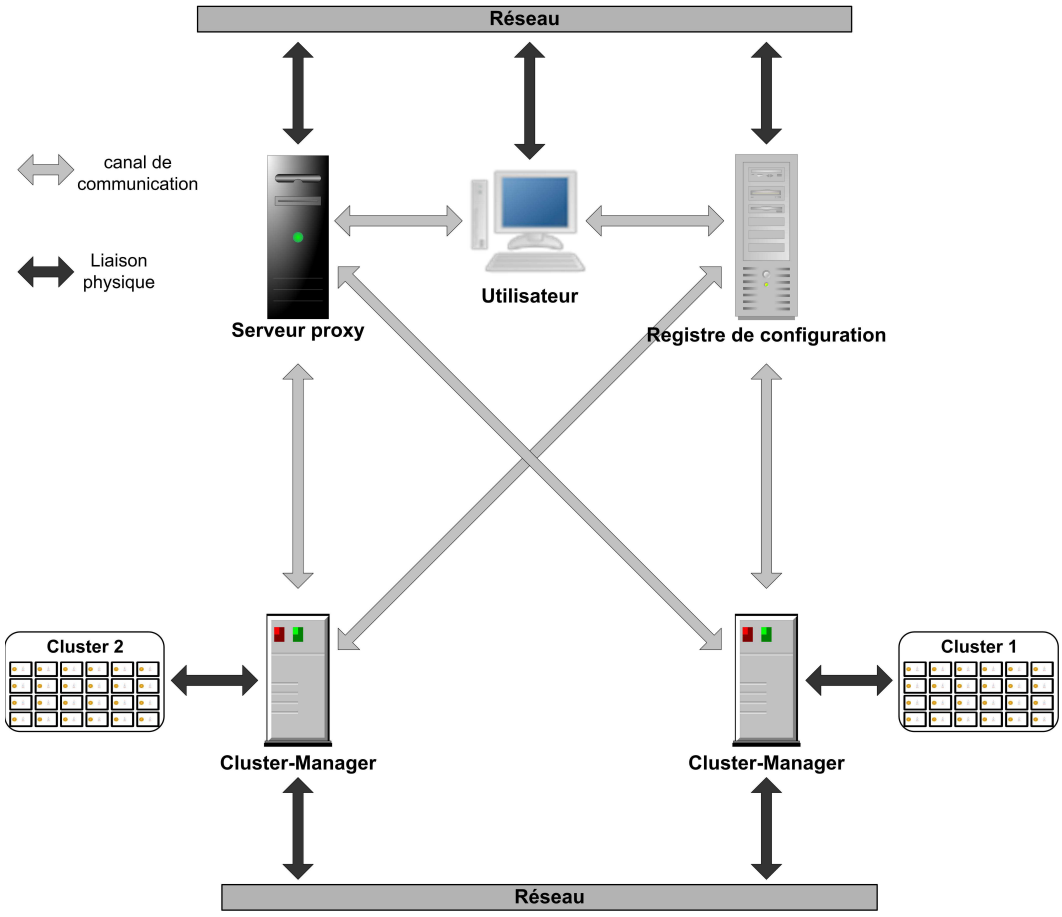


FIG. 5.1 – Les différents composants de la Java Card Grid

Dans cette architecture, nous supposons que le proxy et le registre de configuration possèdent une adresse fixe. Les Cluster-Managers n'ont aucune contrainte de mobilité. Ceux-ci sont dynamiques et

peuvent apparaître et disparaître. De plus, un nouveau Cluster-Manager peut intégrer la grille à tout moment. Un utilisateur doit évidemment pouvoir se connecter depuis n'importe quelle machine.

5.2.2 Les schémas de communication

Le dialogue entre un utilisateur et une carte à puce donnée passe par les différents composants de la grille qui, directement ou indirectement, assurent que les messages sont transmis et acheminés vers la bonne destination. Pour cela, nous définissons différents schémas d'interaction entre les différentes entités de la grille. La communication entre deux parties peut être établie suivant le besoin soit dans les deux sens (symétrique ou bi-directionnelle), soit dans un sens unique (asymétrique ou uni-directionnelle).

La figure 5.2 présente une synthèse des schémas de communication entre les différents composants de la Java Card Grid.

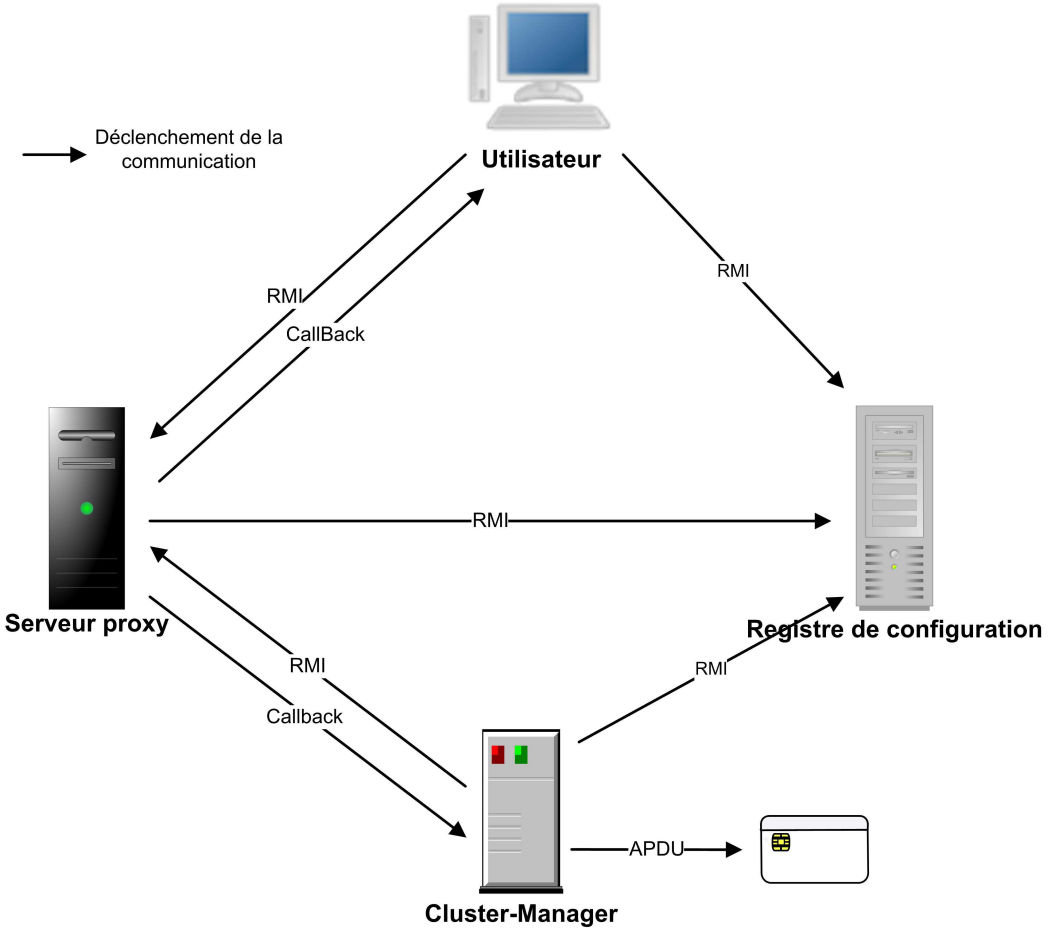


Fig. 5.2 – Les schémas de communication dans la Java Card Grid

Pour communiquer avec une entité dont l'adresse est fixe, nous utilisons la technologie RMI. Alors que lorsqu'il s'agit d'une entité dynamique, nous utilisons une approche de type *Callback*. Quant à elle, la communication avec une carte se fait toujours selon le protocole APDU.

i) Communication entre un utilisateur et le proxy

Comme nous l'avons mentionné, la communication avec une carte à puce passe nécessairement par le proxy, qui joue le rôle de passerelle entre la carte et l'utilisateur. La communication entre le

serveur proxy et l'utilisateur doit être symétrique. En effet, le sens de communication utilisateur[1] vers proxy est utile principalement pour envoyer une commande APDU vers une carte donnée. L'autre sens servira par exemple pour transmettre à l'utilisateur les informations de mise à jour relatives à l'état de la grille (typiquement insertion/arrachage d'une carte).

a) Communication utilisateur vers proxy : afin que les utilisateurs puissent accéder au serveur proxy, nous avons utilisé l'approche RMI où le proxy est le serveur (du point de vue RMI) et les utilisateurs sont les clients. Un utilisateur est donc capable d'initialiser une communication avec le serveur en invoquant les méthodes définies dans l'interface distante de ce dernier.

b) Communication proxy vers utilisateur : la mise en place de ce sens de communication doit prendre en considération les aspects mobilité et dynamisme des clients. Un utilisateur qui s'est déjà connecté depuis une machine quelconque pourra se reconnecter une seconde fois à partir d'une autre machine. Il n'est pas donc possible de lui attribuer une adresse fixe. De plus, nous cherchons à ce que cette communication soit établie sans nécessiter de manipulation de configuration ni au niveau du serveur, ni au niveau du client.

Plusieurs solutions existent permettant d'établir la communication souhaitée. Une des solutions consiste à mettre en place un serveur RMI (RMI Registry). Chaque fois qu'un utilisateur se connecterait à la grille, il devrait fournir son interface distante au serveur RMI. Également lorsqu'il se déconnecterait, il devrait la supprimer. Afin d'éviter d'encombrer notre infrastructure par un serveur supplémentaire, cette solution a été écartée.

Une autre solution, plus élégante, consiste à utiliser les *Callbacks*. Basée sur RMI, cette technique permet au serveur proxy d'appeler des méthodes du client. Ces dernier doit fournir sa référence distante en appelant la méthode appropriée définie dans l'interface distante du serveur proxy grâce à la communication établie dans le premier sens. À travers la référence fournie (qui n'est autre qu'une interface distante), le serveur proxy est capable d'invoquer des méthodes auprès des utilisateurs enregistrés.

La figure 5.3 présente les étapes nécessaires pour établir une communication symétrique entre un utilisateur et le serveur proxy.

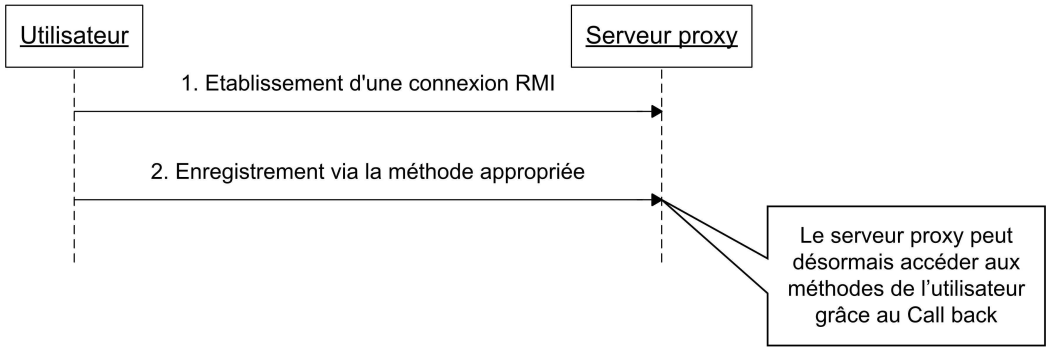


FIG. 5.3 – Mise en place de la communication entre un utilisateur et le serveur proxy

ii) Communication entre le serveur proxy et le Cluster-Manager

Quand le serveur proxy souhaite transmettre une commande APDU à une carte à puce donnée, il doit passer par le Cluster-Manager correspondant. Le proxy doit donc être capable d'engager une communication avec n'importe quel Cluster-Manager. Or, les Cluster-Managers peuvent apparaître et disparaître de façon dynamique. Chaque Cluster-Manager doit donc s'enregistrer auprès du serveur proxy pour que ce dernier puisse disposer de la liste de tous les serveurs de cartes disponibles. Nous utilisons de nouveau l'approche *RMI/Callback* afin d'établir une communication symétrique entre les deux entités. Le schéma RMI classique permet à un Cluster-Manager d'initialiser une communication

[1] En fait le terme "utilisateur" signifie "application côté utilisateur".

avec le serveur proxy, alors que le *Callback* permet d'établir le sens de communication symétrique, c'est-à-dire depuis le serveur proxy vers un Cluster-Manager. Ainsi, de nouveaux Cluster-Managers peuvent intégrer la grille sans qu'aucune reconfiguration ne soit requise. La figure 5.4 illustre le schéma de mise en place de cette communication.

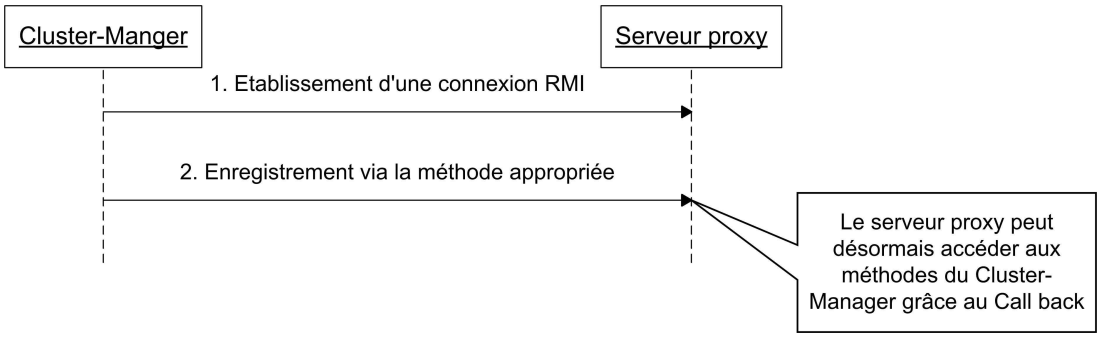


Fig. 5.4 – Mise en place de la communication entre un Cluster-Manager et le serveur proxy

iii) Communication avec le registre de configuration

Le registre de configuration contient les données relatives à l'état de la grille et des cartes. Il est donc souvent sollicité afin de récupérer des informations telles que localisation d'une carte, la description d'un Card Service, etc. De plus, les données enregistrées au niveau du registre de configuration sont régulièrement mises à jour par le proxy ou par un Cluster-Manager. De ce fait, la communication avec le registre de configuration est une communication asymétrique. Le registre doit fournir les moyens permettant d'accéder à ses données. Nous utilisons pour cela une approche de type RMI. Une entité distante pourra alors accéder au registre de configuration afin de mettre à jour ou de consulter les informations qui y sont stockées.

iv) Communication avec la carte à puce

À cause de la passivité des cartes, la communication entre la carte à puce et son Cluster-Manager est réalisée en mode asymétrique. En effet, de par sa conception, une carte à puce est incapable d'engager un dialogue avec le terminal auquel elle est connectée ; elle est passive. Sa fonction se limite à retourner des réponses aux commandes qu'elle reçoit. La communication avec la carte est effectuée suivant le protocole APDU défini dans la partie 4 de la norme ISO-7816. Nous présentons par la suite dans la section 5.5.2 une solution permettant de rendre la carte à puce pro-active, c'est-à-dire capable d'engager une communication avec l'extérieur.

5.3 Le noyau logiciel : présentation générale

Le noyau logiciel de la plate-forme est composé de différents modules imbriqués permettant une utilisation simple et efficace de la grille. Ces composants logiciels fournissent les fonctions de base à travers lesquelles des outils de plus haut niveau (tels que des outils d'administration de la grille, de déploiement d'applications, etc.) pourront être développés. Le noyau logiciel fournit également un support pour accéder aux cartes disponibles et exécuter les applications et les Card Services qui y sont installés. La réalisation du noyau logiciel a nécessité le développement d'environ 2500 lignes de code réparties sur plusieurs classes regroupées dans différents paquetages. Plusieurs versions ont été développées avant d'atteindre la version actuelle.

La figure 5.5 présente l'architecture générale du noyau logiciel. Au niveau inférieur, se trouve le module de contrôle des communications dont le rôle est d'établir des connexions avec les cartes présentes et de créer des canaux de communication avec elles. Le composant de gestion et de contrôle de

l'état de la grille est chargé de repérer les événement qui surviennent, tels que l'insertion et l'arrachage d'une carte, et de réaliser le traitement approprié à la nature de l'événement observé. Parallèlement, on trouve le module d'accès aux cartes dont le rôle est d'établir et de gérer un canal de communication entre une application (distante) et une carte. Ce module assure un rôle complémentaire de celui du gestionnaire des communications. Ce dernier est de bas niveau et est chargé d'établir une communication directe et locale avec la carte selon le protocole APDU. Il fournit des objets proxy à travers lesquels on pourra accéder aux cartes présentes. Le module d'accès aux cartes pour sa part permet de gérer une communication entre une carte, ou plus précisément son objet proxy, et une application distante.

Le module explorateur de cartes permet, en utilisant un jeu de commandes APDU, de récupérer la liste des applications et des Card Services installés sur une carte. Ces informations seront par la suite envoyées au module de publication qui se chargera de les stocker au niveau du registre de configuration.

Ces différents modules logiciels sont déployés sur l'ensemble des composants de la plate-forme Java Card Grid, comme présenté sur la figure 5.5. La gestion des communications avec les cartes est assuré par le Cluster-Manager, grâce à une couche logicielle dédiée. Le contrôle de l'état de la grille, ainsi que l'accès aux cartes font intervenir tous les éléments de la grille, à savoir le serveur proxy, le registre de configuration et le Cluster-Manager. L'explorateur de carte est mis en place au niveau du Cluster-Manager, qui est en liaison directe avec les cartes. La publication de services quant à elle est assurée principalement par le registre de configuration et le Cluster-Manager.

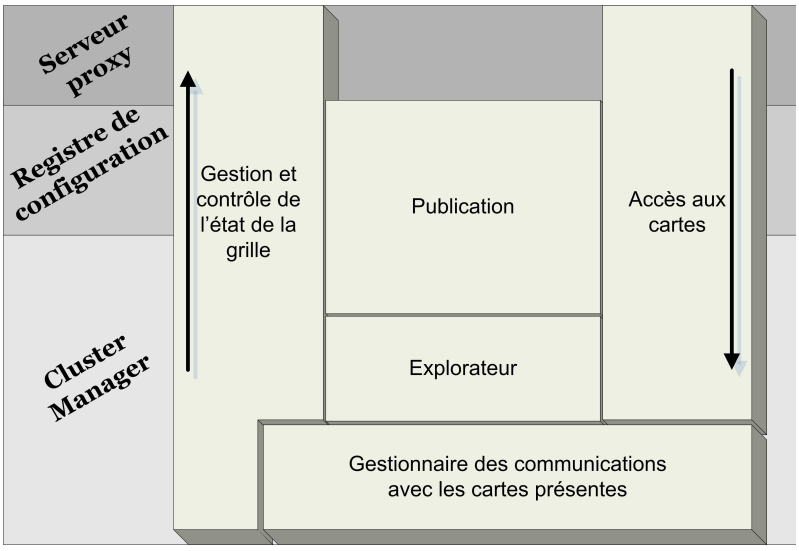


FIG. 5.5 – Les modules logiciels de la plate-forme Java Card Grid

5.4 Gestion des communications avec les Java Cards

Le gestionnaire de communications permet de se connecter aux cartes à puce et d'établir des voies de communication avec elles. Il est composé de deux couches. La première comprend uniquement le *framework* PC/SC [115] , un standard qui permet de communiquer de manière transparente avec la carte via une API dédiée. Une autre possibilité était d'utiliser OCF (*OpenCard Framework*) [112]. Seulement, nous avons préféré la solution PC/SC essentiellement pour sa facilité d'utilisation et sa simplicité. La seconde, de plus haut niveau, fournit des objets proxy à travers lesquels s'effectue les communications avec les cartes.

5.4.1 Le standard PC/SC

Créé en 1996, PC/SC (*Personal Computer/Smart Card*) propose un standard de communication entre un PC et une carte à puce. Les spécifications PC/SC ont été définies en respectant les normes des cartes à puce existantes, en particulier ISO-7816 [84] et EMV [46]. L'avantage principal de PC/SC est qu'il offre une abstraction par rapport aux détails techniques relatifs à la communication avec un lecteur. Ainsi, le développeur d'applications peut se concentrer sur le code métier de son application sans se préoccuper des détails de communication de bas niveau. Le standard PC/SC offre également une API permettant de se connecter à la carte, de lui envoyer des commandes APDU, de récupérer l'état des lecteurs et des cartes, etc.

Un autre avantage de PC/SC est que son architecture est complètement indépendante du système d'exploitation, et qu'il peut donc être implémenté sur n'importe quelle plate-forme. Plusieurs implémentations de PC/SC existent, notamment Windows PC/SC qui est installé par défaut sur la plupart des OS Windows, et PC/SC Lite [116] conçu en particulier pour des plate-formes de type Linux.

Pour utiliser PC/SC en Java, nous avons eu recours à la surcouche JPC/SC [88], une interface JNI (*Java Native Interface*) pour PC/SC.

5.4.2 Gestion des canaux de communication

Au dessus de PC/SC nous avons mis en place une couche logicielle offrant des services de communication de haut niveau. L'accès aux cartes à puce se fait ainsi à travers des objets de type `SmartCardProxy`. Au démarrage du Cluster-Manager, la liste des lecteurs qui sont connectés à la machine est récupérée grâce aux API JPC/SC. Pour chaque lecteur trouvé, une instance de l'objet `ReaderProxy` (listing 5.1) est créée. Si une carte est présente dans le lecteur[2], le `ReaderProxy` crée une instance de l'objet de type `SmartCardProxy` qui permet l'accès à la carte physique. Chaque instance de `SmartCardProxy` se trouve donc rattachée à un objet de type `ReaderProxy`.

```
public class ReaderProxy {

    private Context ctx = new Context();

    SmartCardProxy ScardProxy;

    [...]

     protected ReaderProxy(State state) {


     ctx.EstablishContext(PCSC.SCOPE_SYSTEM, null, null);

        if( (state.dwEventState & PCSC.STATE_PRESENT) == PCSC.STATE_PRESENT ) {

            Card theCard = ctx.Connect(szReader, SHARE_MODE, PROTOCOL_T);

            ScardProxy = new SmatCardProxy(theCard);
        }
     }

[...]
```

Listing 5.1 – La classe `ReaderProxy`

En utilisant les API JPC/SC, la classe `SmartCardProxy` (listing 5.2) offre des primitives de base pour dialoguer avec une carte, telles que la sélection d'une applet ou l'envoi d'une commande APDU. L'objet `SmartCardProxy` fournit également des fonctions de plus haut niveau. Il permet par exemple de récupérer des informations sur la carte, notamment son ATR, son nom, etc.

[2]Les cartes insérées postérieurement sont gérées par le module gestion des événements qui sera présenté dans la section 5.6.

```
public class SmartCardProxy {

   Card card;

   bye[] ATR;

   String name;

   [...]

   protected SmartCardProxy(Card aCard) {

       card = aCard;
       this.identifyCard(); //permet de récupérer les informations d'identification
                            //de la carte tels que son nom, son ATR, etc.

       [...]
    }

    public byte[] sendAPDU(byte[] C_APDU) {

       R_APDU = card.Transmit(C_APDU, 0, C_APDU.length);

       return R_APDU;
    }

    public byte[] selectApplet(byte[] AID) {

       byte[] Select_APDU = build(AID); //permet de construire une commande de sélection
                                        //pour l'AID fourni en paramètre d'entrée

       return card.Transmit(Select_APDU , 0, Select_APDU.length);
    }

   public String getName() {

       return name;
   }

   [...]
```

Listing 5.2 – La classe `SmartCardProxy`

Le système de communication avec la carte, présenté à la figure 5.6, peut être décomposé en deux parties : une partie matérielle et une partie logicielle. La couche matérielle est composée des lecteurs et des cartes physiques. La partie logicielle contient les objets proxy et la couche PC/SC.

5.5 Accès aux cartes

Pour accéder aux cartes, nous avons défini deux modes de communication : un mode passif classique, et un mode pro-actif. Lorsqu'une entité extérieure envoie une commande vers une carte, deux cas de figure peuvent se présenter :

- la carte renvoie une réponse destinée à l'entité qui a envoyé la commande, conformément à une utilisation habituelle. Cette réponse contient le résultat de l'exécution de la commande. Ceci définit le mode passif classique.
- la carte envoie une réponse dans laquelle elle exprime son besoin d'appeler un code extérieur (situé sur une autre carte ou une machine donnée) afin de pouvoir exécuter la commande reçue ; c'est le mode pro-actif.

5.5.1 Le mode passif classique

Comme nous l'avons expliqué précédemment dans la section 5.2.2, nous utilisons une approche de type RMI pour que les utilisateurs puissent accéder à distance aux cartes. Nous avons donc défini l'interface client/serveur `RemoteCardGridProxy`, présentée sur le listing 5.3, qui permet aux utilisateurs

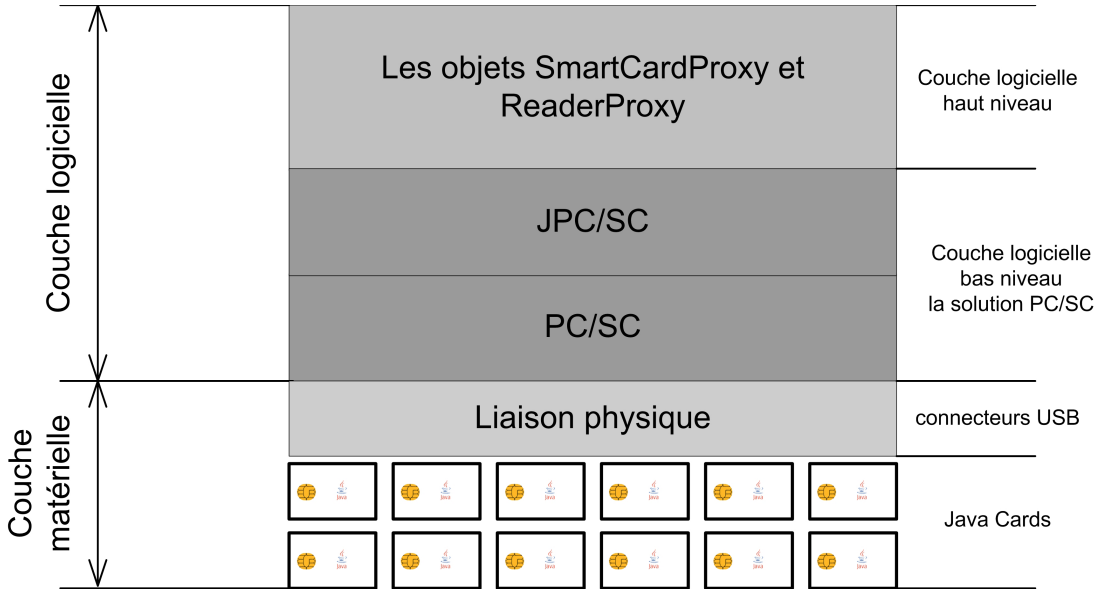


Fig. 5.6 – Le système de communication

d'invoquer des méthodes sur le serveur proxy. À travers les méthodes fournies par cette interface, un utilisateur pourra :

- communiquer avec une carte spécifique en précisant le nom de la carte [3] en question, ainsi que ses données de localisation (nom du lecteur contenant la carte et le Cluster-Manager concerné) ;
- s'enregistrer auprès du diffuseur d'événements pour être informé des changements de l'état de la grille (insertion/arrachage d'une carte, arrêt d'un Cluster-Manager, etc.)

```
public interface RemoteCardGridProxy extends Remote {

    public String subscribe(GridEventListener geventlistener);

    public void unsubscribe(String ref); //ref correspond à la référence retournée par la méthode subscribe

    public byte[] sendAPDU(String theClusterManager, String theReader, String theCard, byte[] C_APDU);

    public byte[] selectApplet(String theClusterManager, String theReader, String theCard, byte[] C_APDU);

    [...]
```

Listing 5.3 – L'interface `RemoteCardGridProxy`

Du point de vue fonctionnel, l'accès aux cartes est réalisé en trois étapes que nous décrivons dans la suite et qui sont présentées à la figure 5.7.

Étape 1 : Localisation d'une carte

Pour accéder à une carte, il faut tout d'abord la localiser. Comme nous l'avons vu dans le chapitre 4, les informations de localisation sont constituées par le triplet <nom du Cluster-Manager où se trouve la carte, nom du lecteur contenant la carte, nom de la carte>.

Pour pouvoir identifier le Cluster-Manager concerné, nous utilisons son adresse IP (ou son nom). Le nom du lecteur de cartes quant à lui est géré par le standard PC/SC. PC/SC attribue un même nom pour deux lecteurs identiques. Pour avoir des noms différents pour des lecteurs de même type, nous avons dû apporter quelques modifications aux sources PC/SC. Ceci s'avère utile lorsqu'on s'intéresse

[3] Un pseudonyme qui permet d'identifier les cartes de façon unique appelé aussi *CardIDentifier* (CID).

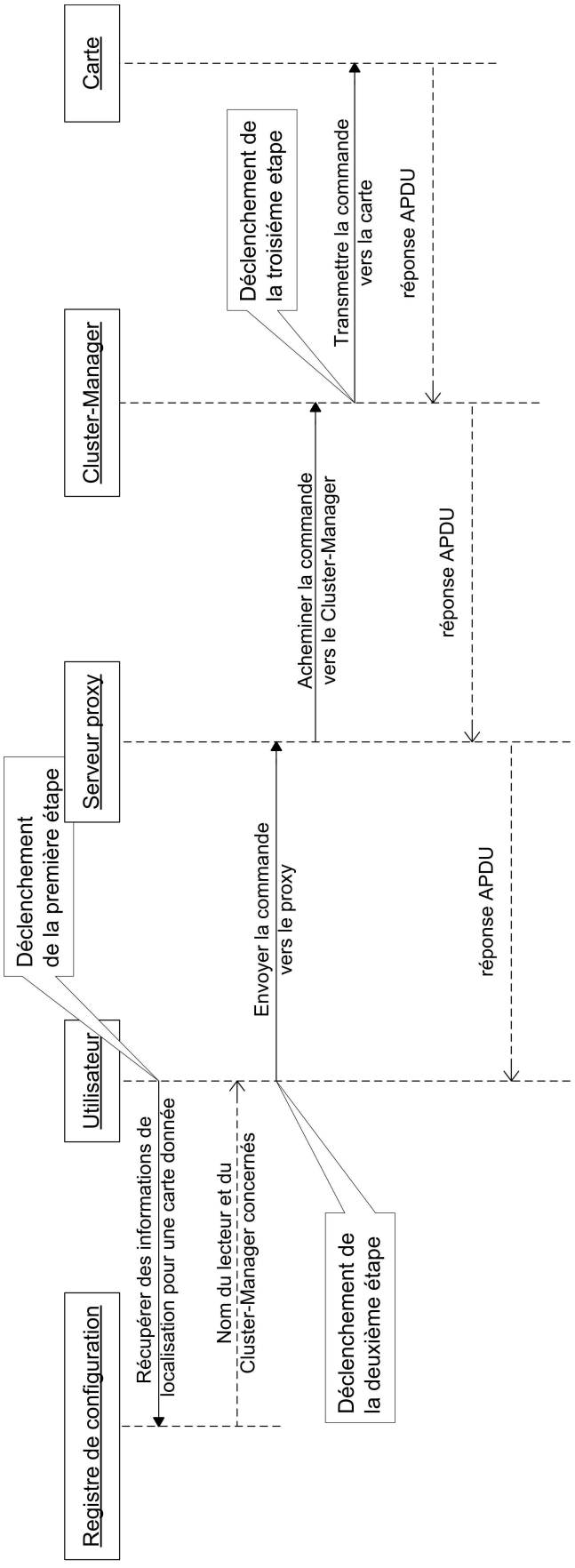


FIG. 5.7 – Mise en place des étapes nécessaires pour accéder à une carte

aux lecteurs, notamment pour un administrateur afin de vérifier dans quel état se trouve un lecteur donné. Dans le listing 5.4, nous présentons ce qui constitue notre contribution à PC/SC[4] : le nom du lecteur et changé en ajoutant le numéro de série du lecteur. Grâce à cette modification, nous avons pu mettre en place un outil d'administration temps réel, décrit dans l'annexe B.

```
/*
 * MUSCLE SmartCard Development ( http://www.linuxnet.com )
 *
 * $Id: hotplug_libusb.c $
 *
 */

static struct _readerTracker {
    char status;
    char bus_device[BUS_DEVICE_STRSIZE]; /* device name */
    char *fullName; /* full reader name (including serial number) */
} readerTracker[PCSCLITE_MAX_READERS_CONTEXTS];


LONG HPAddHotPluggable(struct usb_device *dev, const char bus_device[], struct _driverTracker *driver) {

[...]

#ifdef ADD_SERIAL_NUMBER
    if (dev->descriptor.iSerialNumber)
    {
        usb_dev_handle *device;
        char serialNumber[MAX_READERNAME];
        char fullname[MAX_READERNAME];

        device = usb_open(dev);
        usb_get_string_simple(device, dev->descriptor.iSerialNumber, serialNumber, MAX_READERNAME);
        usb_close(device);

        snprintf(fullname, sizeof(fullname), "%s␣(%s)", driver->readerName, serialNumber);

        readerTracker[i].fullName = strdup(fullname);
    }
    else
#endif
        readerTracker[i].fullName = strdup(driver->readerName);

[...]
```

Listing 5.4 – Modification de l'attribution des noms de lecteurs dans PC/SC Lite

L'identifiant de la carte quant à lui est géré par une application embarquée. Nous avons en effet réalisé une applet Java Card permettant de nommer (ou même de renommer) et de récupérer l'identifiant de la carte. Ainsi, il est possible de distinguer les différentes cartes par leur nom. Afin d'obtenir les informations de localisation complètes, il suffit d'envoyer au registre de configuration une requête en précisant l'identifiant de la carte recherchée.

Étape 2 : Transmission des requêtes

L'objectif de cette étape est d'acheminer la requête vers le bon Cluster-Manager. Pour cela, l'utilisateur doit tout d'abord appeler la méthode `sendAPDU` (définie dans l'interface `RemoteCardGridProxy`) sur le serveur proxy en précisant les noms du Cluster-Manager et du lecteur obtenus dans l'étape 1. Par la suite, en utilisant la technique de *Callback*, le serveur proxy transmet la requête de l'utilisateur vers le Cluster-Manager auquel la carte est connectée. Cette communication est établie en en invoquant la méthode `sendAPDU` sur le serveur de cartes. Cette méthode étant définie dans l'objet distant `ClusterChannel` (listing 5.5), fourni par le Cluster-Manager lors de la phase d'enregistrement auprès du serveur proxy.

[4]Cette modification ne concerne que le projet PC/SC Lite, qui est en *open source* (BSD License).

```
public interface ClusterChannel extends Remote {

   public byte[] sendAPDU(String theReader, String theCard, byte[] C_APDU);

   public byte[] selectApplet(String theReader, String theCard, byte[] AID);

   [...]
```

Listing 5.5 – L'objet de référence du Cluster-Manager

Étape 3 : Envoi de la commande APDU

Une fois que la requête de l'utilisateur a été acheminée vers le bon Cluster-Manager, la commande APDU peut être envoyée à la carte. Le Cluster-Manager est capable d'identifier l'objet `ReaderProxy` concerné (qui correspond au lecteur dans lequel la carte est insérée) grâce au nom de lecteur fourni comme paramètre d'entrée à la méthode `sendAPDU`. Ensuite, à travers le `SmartCardProxy` du `ReaderProxy`, la commande APDU est envoyée à la carte. La réponse retournée par la carte sera alors transmise vers l'utilisateur sous forme d'un tableau d'octets en suivant le chemin inverse du chemin d'appel.

5.5.2 Le mode pro-actif

De par sa conception, la carte à puce est une entité passive. La norme ISO-7816 [84] ne prévoit pas de mécanisme lui permettant d'invoquer un code extérieur. Une solution à ce problème a par exemple été introduite dans [91] et consiste en la mise en place d'une architecture appelée AWARE permettant à la carte d'être réactive vis-à-vis de son environnement extérieur et donc de pouvoir émettre des requêtes destinées à des applications distantes. Son rôle passe donc d'un simple serveur à un véritable client qui peut demander l'exécution d'un service. Cette architecture est basée sur trois composants comme présenté à la figure 5.8 :

- La carte qui contient une application encartée et qui a besoin d'un service extérieur pour pouvoir exécuter la commande reçue. Grâce à l'architecture, la carte serait capable d'émettre une requête à destination d'une application distante.
- Le logiciel situé sur le terminal auquel la carte se trouve connectée. Il assure une fonction de passerelle entre la carte et l'application distante. Il est composé de deux parties :
  - une partie générique : c'est la partie logicielle constituée des éléments communs à toutes les applications encartées et qui est indépendante d'un type d'application particulière. Il s'agit par exemple des modules pour la gestion et le routage des requêtes entrantes et sortantes et qui doivent être présents en permanence sur le terminal ;
  - une partie dédiée : c'est la partie logicielle constituée des briques supplémentaires spécifiques à l'application distante et nécessaires pour son exécution. Parmi les constituants de cette partie, nous trouvons par exemple des éléments d'adaptation qui sont des représentants de l'application distante ce qui permet de réaliser l'invocation de ses méthodes.
- L'environnement extérieur représente un système d'information réparti et englobe :
  - les applications distantes ;
  - un référentiel des éléments d'adaptation qui permet l'installation sur le terminal de la partie dédiée ;
  - une entité de localisation incluant un service de nommage permettant de trouver les applications distantes. En effet, pour pouvoir invoquer un service, il faut tout d'abord localiser le serveur où le service se trouve.

À travers cette architecture, la carte peut interagir avec son milieu extérieur et invoquer des applications distantes ce qui permet son intégration dans des systèmes répartis.

Pour notre part, nous proposons une solution dans laquelle une carte, suite à une commande APDU, renvoie une réponse dans laquelle elle exprime son besoin d'exécution d'un code extérieur. Un PC intermédiaire (jouant le rôle de routeur) traite la requête de la carte (encapsulée dans sa réponse)

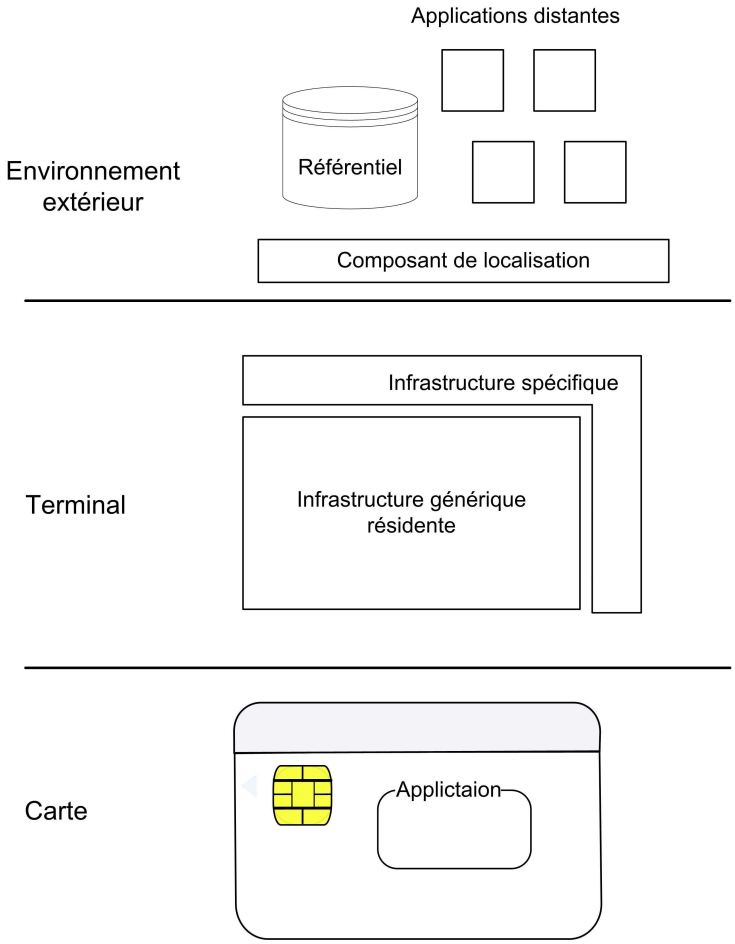


Fig. 5.8 – L'architecture AWARE

et l'envoie vers la destination indiquée par la carte. Nous distinguons deux cas pour la proactivité, la proactivité simple et la proactivité forte.

5.5.2.1 Proactivité forte

Le mode de proactivité forte consiste à ce qu'une carte cliente (appelée aussi carte émettrice) puisse exécuter une application se trouvant sur une autre carte distante. Le dialogue entre les deux parties s'effectue grâce à un routeur qui permet d'établir un canal de communication virtuel entre les deux cartes. Par ailleurs, il est important de préciser que dans le projet Java Card Grid, nous supposons que si deux applets ont le même AID, alors il s'agit nécessairement de la même application (même code). Il suffit donc au routeur de trouver une carte (appelée carte cible) contenant une applet dont l'AID est celui spécifié par la carte cliente. Les étapes nécessaires à la réalisation de la proactivité forte, représentées à la figure 5.9, consistent en :

1. sélection d'une applet distante : afin d'engager une communication pro-active, une carte doit tout d'abord construire une réponse, dite réponse de sélection d'applet extérieure, dans laquelle elle précise l'AID de l'applet à sélectionner. Ce type de réponse se distingue des autres par des valeurs particulières des mots d'état (SW1=97, SW2=A4). Le champ de données quant à lui contient l'AID de l'applet cible.
2. lorsque le Cluster-Manager reçoit de la carte émettrice (source) une réponse de sélection d'une applet extérieure, deux cas de figure peuvent se présenter (figure 5.10) :

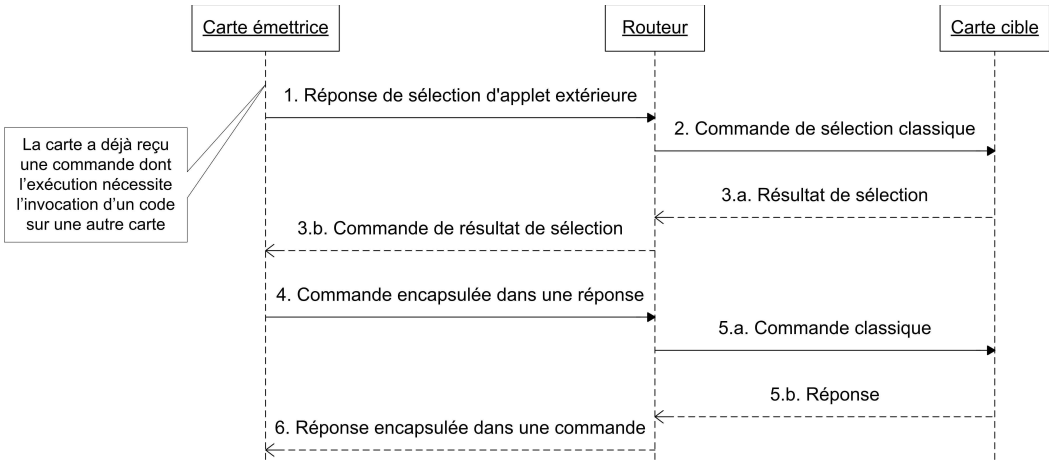


FIG. 5.9 – Étapes de mise en œuvre de la proactivité forte

– s'il trouve une carte appropriée dans l'ensemble des cartes qu'il gère, il établit alors un canal de communication avec celle-ci et se charge de la fonction de routage,
– sinon, il transfère la réponse, qui contient toujours une requête de sélection, vers le serveur proxy qui essaie de trouver une carte candidate et prend donc le rôle du routeur.

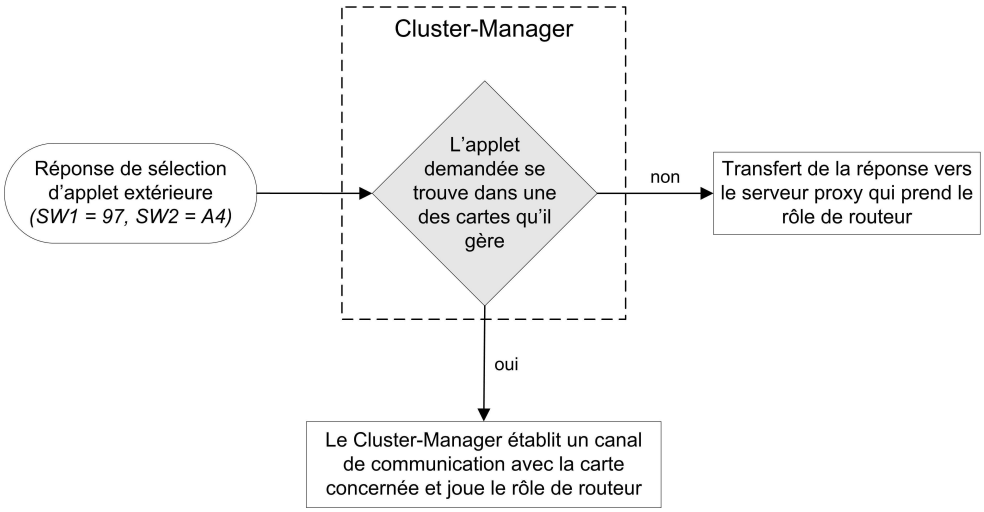


FIG. 5.10 – Manière de désignation du routeur

Muni des données contenues dans la réponse de sélection, le routeur construit une commande de sélection classique qui est envoyée vers la carte cible (destination).

3. après avoir reçu une réponse de la carte cible, le routeur construit une nouvelle commande à envoyer à la carte cliente. Il encapsule la réponse qu'il vient de recevoir dans le champ de données de cette commande. L'en-tête de cette dernière (CLA, INS, P1, P2) est identique à celui de la commande envoyée par l'utilisateur et ayant déclenché le mode procatif.
4. une fois que l'applet distante est sélectionnée, la carte émettrice construit une nouvelle réponse dont le champ de données contient une commande APDU. . Les mots d'état doivent être mis à des valeurs particulières (SW1=97, SW2=xx).
5. le routeur envoie alors la commande vers la carte cible qui lui retourne une réponse.

6. le routeur place cette réponse dans le champ de données d'une nouvelle commande APDU et l'envoie vers la carte source. L'entête de cette commande correspond toujours à l'entête de la commande qui été envoyée initialement par l'utilisateur.

Problème de reprise de code

Origine du problème : la mise en place du mécanisme de la proactivité forte engendre un problème de reprise de code. En effet, selon le modèle que nous avons décrit, la résultat de sélection de l'applet distante est transmise vers la carte émettrice dans une commande dont l'entête est identique à celui envoyé initialement par l'utilisateur. Ceci reste également vrai pour la réponse d'exécution du code extérieur. Le fait d'envoyer des commandes successives avec le même entête (et en particulier le même valeur du champ instruction) conduit à exécuter infiniment un même bloc de code sans issue.

Explication : une applet possède un point d'entrée unique : la méthode *process()*. Chaque commande entrante entraîne systématiquement l'invocation de la méthode process sur l'applet sélectionnée. Or la partie de code à exécuter dépend du champ de l'instruction de la comamnde[5]. Dès lors, le fait d'envoyer deux commandes, ou plus, possédant la même valeur d'instruction provoque l'exécution du même bloc. Dans le modèle la proactivité, trois commandes portant la même valeur d'instruction sont envoyées vers la carte, ce qui implique reprendre l'exécution de la même partie de code et donc de boucler à l'infini.

La solution : afin que la carte source puisse redémarrer sur la ligne de code suivant immédiatement l'envoi d'une réponse de type pro-actif, nous proposons l'utilisation d'une variable de repère. Cette variable sera affectée à une valeur qui correspond au bloc qui sera exécuté à la réception de la prochaine commande APDU, c'est-a-dire la commande transportant la réponse de la requête pro-active ayant provoquée la sortie de la méthode courante. En utilisant la structure *switch*, il sera possible de déterminer facilement le point de reprise comme illustré par le listing 5.6. À la fin d'une session pro-active, la variable sera réinitialisée à sa valeur par défaut.

Les différentes états associés à une applet lors d'une session proactive sont présentées sur la figure 5.11. Nous représentons également les transitions qui peuvent exister entre ces différents états.

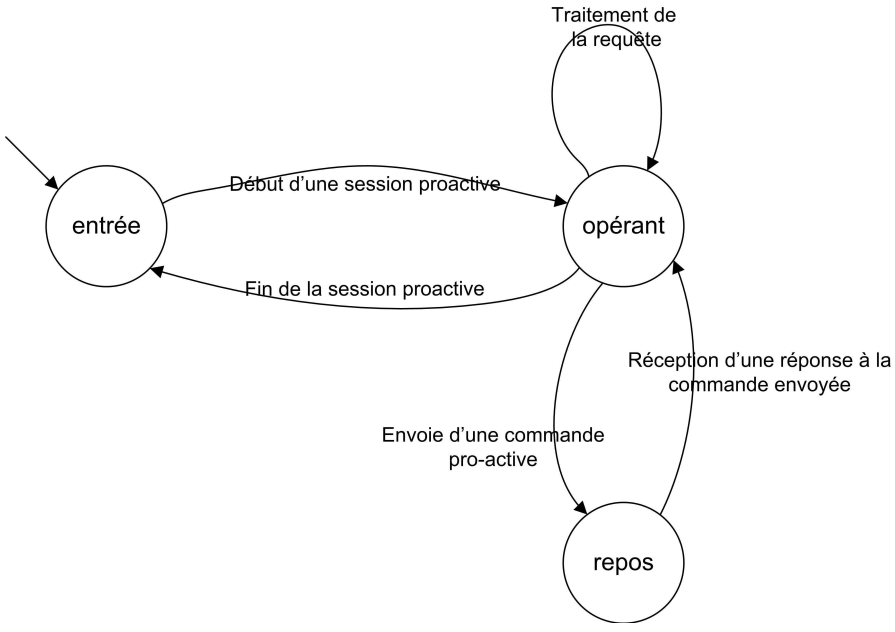


Fig. 5.11 – États et changements lors d'une session proactive

[5]En effet, le champ instruction permet de préciser la valeur d'instruction, les champs P1 et P2 servent à paramétrer l'instruction.

```
/*****
* Traitement dépendant de la variable de repère token
*
****/
switch (token)

    /**
    * Point d'entrée classique
    *
    */
    case INITIAL_VALUE :

        /**
        * Traitements à effectuer
        *
        */

        /**
        * Construction d'une requête de sélection d'applet extérieure : SW1 = 0x97, SW2 = 0xA4
        *
        */

        /**
        * Mémorisation de l'envoi d'une réponse de sélection et affectation de token à la valeur appropriée
        *
        */
        token = AFTER_SELECTION

       /**
        * Sortir de la boucle et de la méthode pour renvoyer la réponse
        *
        */
       break;

    /**
    * Point de reprise après réception de la réponse de la demande de sélection
    *
    */
    case AFTER_SELECTION :

        /**
        * Vérification de la réussite de la sélection, et éventuellement des traitements à effectuer
        */

        /**
        * Construction d'une requête pro-active, encapsulée dans une réponse : SW1 = 0x97, SW2 = xx
        *
        */

        /**
        * Mémorisation de l'envoi d'une réponse pro-active et affectation de token à la valeur appropriée
        *
        */
        token = AFTER_COMMAND

       /**
        * Sortir de la boucle et de la méthode pour renvoyer la réponse
        *
        */
       break;

    /**
    * Point de reprise après réception du résultat
    *
    */
    case AFTER_COMMAND :

        /**
        * Récupération du résultat de l'exécution de la commande pro-active, et éventuellement des traitements à effectuer
        *
        */

        /**
        * Mémorisation de la fin de la session pro-active et affectation de token à la valeur appropriée
        *
        */
        token = INITIAL_VALUE;

       /**
        * Sortir de la boucle pour finir les traitements
        *
        */
       break;
```

Listing 5.6 – Le mécanisme de reprise de code

5.5.2.2 Proactivité simple

À la différence avec le mode fort, la proactivité simple permet à une carte d'appeler un code se trouvant sur une machine donnée, et non pas sur une autre carte. Dans ce mode de proactivité, le serveur proxy doit être configuré pour qu'il puisse traiter des réponses/requêtes de ce type. En effet, lorsqu'une carte souhaite appeler un code se trouvant sur une machine distante, elle doit construire une réponse APDU dans laquelle elle met les mots d'état à des valeurs déterminées permettant d'identifier l'application extérieure à exécuter. Lorsque le serveur proxy reçoit une réponse de ce type, il essaie de retrouver le programme, appelé programme passerelle, qui permet d'appeler l'application extérieure en se référant aux informations enregistrées dans un fichier de configuration. Celui-ci contient une correspondance entre les mots d'état et les programmes passerelles installés sur le serveur proxy. Ces programmes permettent d'accéder aux applications se trouvant sur des machine distantes.

Après exécution de l'application distante, le serveur proxy récupère (via un fichier qui sert de tampon pour les messages APDU) une commande APDU et la transmet à la carte. Cette procédure générique d'appel d'application distante permet d'ajouter de nouveaux programmes passerelles sans avoir besoin de remanier le code au niveau du serveur proxy. La figure 5.12 présente les étapes d'une session pro-active simple.

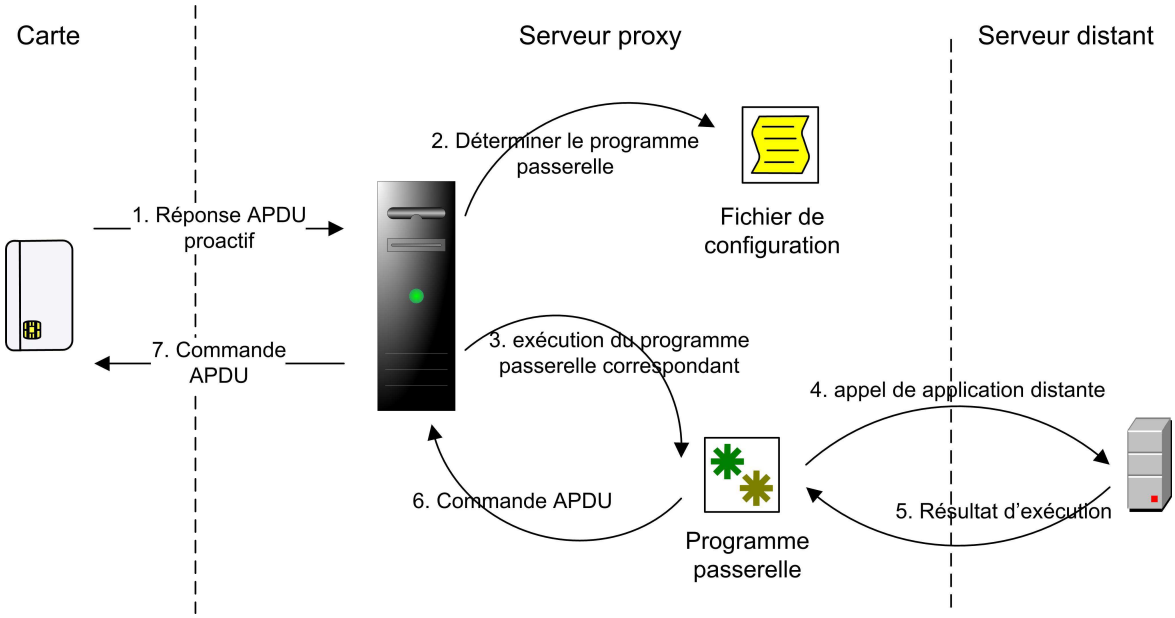


Fig. 5.12 – La proactivité simple

Dans le chapitre 6, nous présentons une utilisation de ce mode de proactivité dans le cadre de la sécurité des communications, et en particulier pour la vérification de certificats utilisateurs.

5.6 Contrôle et gestion de l'état de la grille

Le système de contrôle de l'état de la grille offre les moyens pour déterminer à tout moment les cartes qui sont réellement présentes dans la grille. La construction d'une vue de la structure globale de la grille est indispensable pour en assurer une administration efficace. Par ailleurs, la connaissance des cartes présentes dans la grille nous permet d'éviter, ou au moins de limiter, l'envoi de commandes APDU vers des cartes qui ont été arrachées, et par conséquent ne sont plus accessibles. La construction et la gestion de la structure globale de la grille sont réalisées en deux phases :

– la première phase concerne les serveurs de cartes. Pour pouvoir intégrer la grille, un ClusterManager doit s'enregistrer (via la méthode `subscribe`) auprès du serveur proxy en fournissant sa référence distante. Cette méthode est définie dans l'interface `RemoteClusterManager` (listing

5.7), dont la réalisation se fait au niveau du serveur proxy. De plus, le Cluster-Manager doit également récupérer l'état initial des lecteurs forment la cluster et effectuer l'exploration des cartes disponibles (grâce à l'explorateur de carets présenté dans la section 5.7) afin d'obtenir la liste des applications et des Cards Services installés sur les cartes. Ces informations sont envoyées vers le registre de configuration pour être publiées.
Il faut préciser que lorsqu'un Cluster-Manager intègre la grille, le serveur proxy signale aux utilisateurs intéressés la connexion d'un nouveau serveur de cartes afin de consulter le registre de configuration si besoin. La diffusion des événements se fait via les méthodes définies dans l'interface de notification `GridEventListener` fournie par le client lors de la phase d'enregistrement (voir section 5.5.1). C'est la technique de *Callback* qui est utilisée lors de cette communication.
De façon réciproque, lorsqu'un Cluster-Manager veut se déconnecter de la grille, il invoque la méthode `unsubscribe` sur le serveur proxy qui se charge d'informer le registre de configuration pour que ce dernier met à jour ses données. Les utilisateurs enregistrés sont également mis au courant de cette déconnexion.

– la deuxième phase concerne l'état des lecteurs et des cartes. Après avoir récupéré l'état initial des lecteurs formant le cluster, le Cluster-Manager doit être capable de déterminer les éventuels changements d'état qui peuvent se produire au niveau des cartes et/ou des lecteurs. Chaque fois qu'il y a un changement d'état, le registre de configuration et le serveur proxy doivent être informés. Le serveur proxy pourra alors diffuser l'information aux utilisateurs intéressés de la même façon que dans la phase 1. Nous détaillons dans ce qui suit les fonction de gestion des événements qui concernent les lecteurs et les cartes.

```
public interface RemoteClusterManager extends Remote {

    public void subscribe(ClusterChannel remoteRef, String hostName);

    public void unsubscribe(String hostName);

[...]
```

Listing 5.7 – L'interface `RemoteClusterManager`

Gestion des événements liés aux lecteurs/cartes

La gestion de l'état des lecteurs et des cartes est basée sur la détection des événements qui peuvent survenir. Cette étape est fondamentale puisqu'elle permet d'avoir une vue conforme à la structure réelle de la grille. Les événements survenant au niveau des lecteurs et des cartes sont interceptés par un *listener* dédié. Nous avons à ce propos développé un *ListernerEvent* qui est installé sur chaque Cluster-Manager permettant de contrôler les lecteurs et les cartes, et de repérer tout changement de leur état. Du point de vue fonctionnel, il s'agit d'une classe héritant de `Thread` et qui tourne en arrière plan. Dans cette classe nous faisons appel à la fonction `GetStatusChange` définie dans les API JPC/SC. Cette méthode constitue un appel bloquant jusqu'à ce que l'état de l'un des lecteurs connectés au Cluster-Manager change. Cependant elle ne fournit aucune information concernant l'emplacement et la nature de l'événement survenu. Pour cela, nous avons implémenté le code nécessaire (la méthode *getStatutsChange()*) afin de récupérer l'endroit exact (c'est-à-dire au niveau de quel lecteur l'événement s'est produit). Cette méthode parcourt un tableau stockant l'état (objet `State`) de chaque lecteur afin de trouver celui où l'événement s'est produit. Une fois trouvé, la ligne de commande `SCardGridManager.getStatusChange (stateReaders[i].szReader, dwOldState, stateReaders[i].dwEventState)` permet de préciser la nature de l'événement (par exemple insertion ou arrachage d'une carte) comme illustré dans le listing 5.8.

Par la suite, un traitement approprié à la nature de l'événement identifié va être réalisé. Nous nous intéressons principalement aux événements qui conduisent au changement de la structure globale de

```
public class ListenerEvent extends Thread {

     /**
     * Définition d'un tableau stateReaders dont les élément représentent l'état de chaque lecteur.
     *
     */
    State[] stateReaders;

    Context ctx = new Context();

    [...]

    public void run(){

        [...]
        /**
        * Instruction bloquante. Attendre jusqu'à changement de l'état d'un lecteur
        *
        */
        ctx.GetStatusChange(PCSC.INFINITE, stateReaders);

        /**
        * Déterminer le lecteur correspondant
        *
        */
        this.getStatusChange ();

        run();
        }


     /**
    * Cette méthode permet de fournir des informations quant à la localisation et la nature de l'événement produit
    *
    */
    private void getStatusChange () {


        /**
        * Parcourir le tableau stateReaders pour trouver le lecteur où l'événement s'est produit
        *
        **/
        for ( int i = 0; i < stateReaders.length; i++) {

            if ( (stateReaders[i].dwEventState & PCSC.STATE_CHANGED) == PCSC.STATE_CHANGED ) {
                int dwOldState = stateReaders[i].dwCurrentState;

                stateReaders[i].dwCurrentState = stateReaders[i].dwEventState;


                /**
                * pour identifier la nature de l'événement
                *
                */
                SCardGridManager.getStatusChange (stateReaders[i].szReader, dwOldState, stateReaders[i].dwEventState);

            }
            else
                continue;
        }
    }

 [...]
}
```

Listing 5.8 – L'écouteur d'événement

la grille, c'est-à-dire insertion ou arrachage d'une carte. Quand un événement de ce type se produit, les informations au niveau du registre de configuration sont mises à jour :

- s'il s'agit d'un événement qui a pour effet l'apparition d'une carte, la liste des applications et des Card Services qui y sont installés est récupérée par l'explorateur de cartes (qui sera présenté dans la section 5.7). Les informations obtenues sont ensuite envoyées au registre de configuration pour être publiées.

– si l'événement conduit à la disparition d'une carte, toutes les données dynamiques (telles que le contenu de la carte) qui la concernent sont supprimées du registre de configuration.

Il existe d'autres événements qui n'ont pas un effet sur la structure globale de la Java Card Grid, mais plutôt sur l'état propre d'une carte. Par exemple, une carte qui était en communication, devient disponible, ou l'inverse. Ces événements sont pour l'instant ignorés : aucun traitement n'est déclenché. Cependant, ce type d'événement pourrait servir à réaliser des mesures statistiques (calcul du temps d'utilisation d'une carte, par exemple). La figure 5.13 présente le mode de fonctionnement du gestionnaire d'événements.

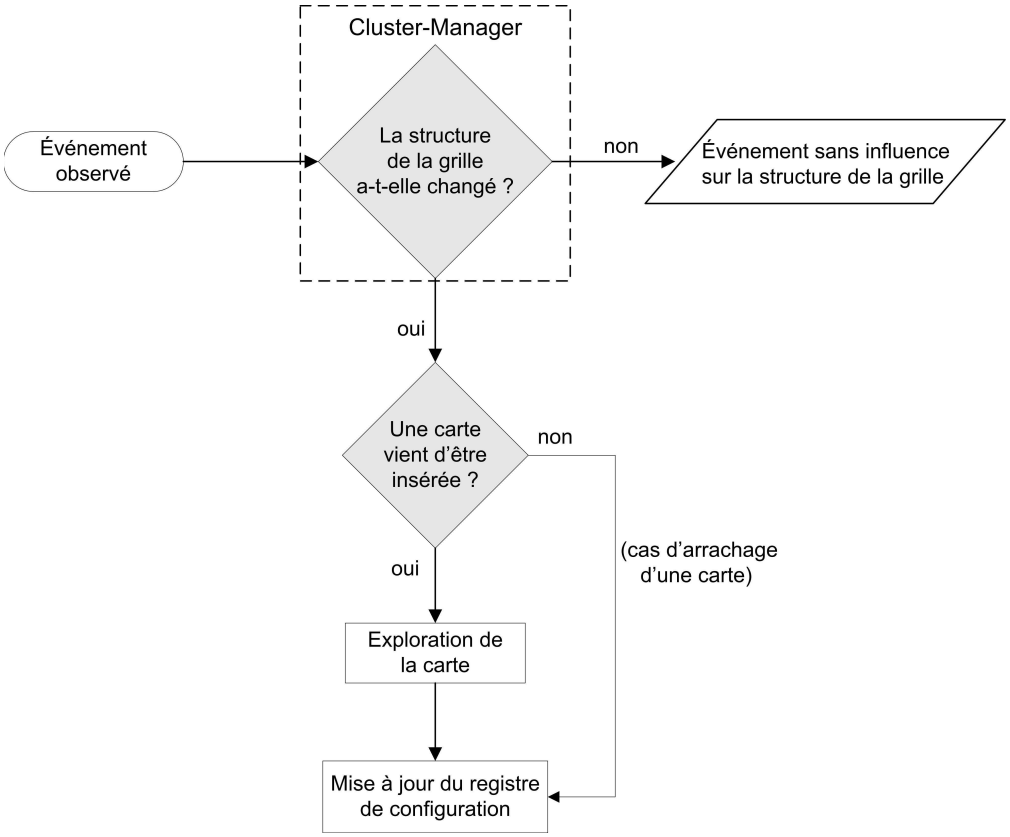


FIG. 5.13 – Le gestionnaire des événements liés aux lecteurs/cartes

Finalement, les utilisateurs sont informés de l'occurrence des événements via les méthodes définies dans l'interface de notification fournie au serveur proxy dans la phase d'enregistrement.

5.7 Exploration et publication du contenu de la carte

L'explorateur de cartes permet de récupérer le contenu d'une carte donnée, c'est-à-dire la liste des AID des applets et la description des Card Services qui y sont installés. Cette fonctionnalité peut être réalisée selon les deux modes suivants :

exploration brute : ce mode d'exploration permet d'obtenir la liste des AID des applications chargées. Pour cela, nous utilisons la commande APDU *Get Status* définie dans les spécifications GlobalPlatform. Les différentes variantes de cette commande permettent de retourner la liste des AID des applications installées sur la carte. En plus de l'AID, la commande *Get Status* permet de retourner le type de l'application : application exécutable ou non (applet ou package), domaine de sécurité, etc.

exploration spécifique : ce mode d'exploration ne peut être effectué qu'après vérification de la présence de l'applet *CatalogService*. Cette application joue le rôle d'un annuaire encarté. Elle maintient la liste des applets qui ont été déclarées comme des Card Services. L'applet *CatalogService* permet de retourner les descriptions des Card Services installés. Seule la description fonctionnelle est récupérée. Il s'agit de la partie écrite par le fournisseur de l'application et chargée sur la carte au moment de l'installation de l'application comme nous l'avons expliqué dans la section 4.6.2.2. Une exemple de description fonctionnelle est donné par le listing 5.9.

```
<applet name="BasicCalcul" id="0x0A0B0C0D112233">
    <method>
        <name>multiplyNumbers</name>
        <ins>02</ins>
        <arg>short</arg>
        <arg>short</arg>
        <type>short</type>
        <documentation>Effectuer la multiplication de deux nombres de type short</documentation>
    </method>

  [...]

</applet>
```

Listing 5.9 – Exemple de description fonctionnelle d'un Card Service

Une fois le phase d'exploration terminée, les informations récupérées sont publiées à travers le registre de configuration. Ce dernier doit également contenir les données de localisation de chaque carte, afin qu'un utilisateur puisse y accéder. Ainsi, en consultant le registre de configuration par exemple pour la recherche d'un service donné, un utilisateur récupère la description complète correspondante dont le schéma général est similaire à celui présenté sur le listing 5.10.

```
<!-- Niveau Localisation -->

<Localized>

    <Carte>Card_IDentifier</Card>
    <reader>Name_Reader</reader>
    <ClusterManager>Name_Host</ClusterManager>

</localized>

[...]

<!-- Niveau technique -->

<applet name="BasicCalcul" id="0x0A0B0C0D112233">
    <method>
        <name>multiplyNumbers</name>
        <ins>02</ins>
        <arg>short</arg>
        <arg>short</arg>
        <type>short</type>
        <documentation>Effectuer la multiplication de deux nombres de type short</documentation>
    </method>

    [...]

</applet>
```

Listing 5.10 – Exemple de description complète d'un Card Service

On notera que les informations concernant le contenu de la carte peuvent être sauvegardées au niveau du Cluster-Manager qui pourra s'en servir lors d'une communication dans le mode pro-actif que nous avons présenté dans la section 5.5.2.1. Ces données peuvent être enregistrées par exemple sous forme d'un document XML comme illustré sur le listing 5.11.

```
<carte>JavaCard</carte>


<application>

   <AID>0x0A0B0C0D112233</AID>
   <type>applet</type>
   <is service>yes</is service>

</application>

[...]
```

Listing 5.11 – Exemple de description du contenu d'une carte sous format XML

5.8 Conclusion

Dans ce chapitre, nous avons présenté un modèle d'implémentation pour de la couche intergiciel d'une grille de cartes à puce de type Java. Nous avons également décrit les principales fonctionnalités fournies par cette couche. Toutefois, la sécurité des communications n'a pas été traitée : ce sera l'objet du chapitre suivant.

Chapitre 6

Approche basée PKI pour la sécurisation des échanges

Sommaire

6.1 Introduction

La sécurité des communications est un problème commun à toutes les architectures distribuées. Les messages échangés entre les différents nœuds du système peuvent être la cible de plusieurs sortes d'attaques [17] (usurpation d'identité, rejeu, *man in the middle*, etc.) mettant en cause la sécurité des communications. Afin de se prémunir contre ces menaces, des procédures sécuritaires doivent être mises en place permettant de garantir les propriétés de sécurité générales telles que la confidentialité des données ou l'intégrité des messages.

Dans ce chapitre, nous décrivons la solution que nous avons conçue pour sécuriser les communications entre les cartes et leurs utilisateurs dans le contexte de la grille de cartes à puce. La solution proposée a été validée formellement à l'aide de l'outil AVISPA [7].

6.2 Étude de la problématique

Dans une grille de cartes à puce, chaque carte est considérée comme un nœud autonome accessible par plusieurs utilisateurs. Les canaux de communication établis avec elle doivent donc être sécurisés. Le grand défi est de concevoir un protocole de sécurité qui soit à la fois efficace et simple, afin de ne pas dégrader les performances du système vues les ressources limitées des cartes à puce. Notre première proposition a été d'utiliser les protocoles définis dans GlobalPlatform [67]. Des implémentations de cette spécification sont en effet présentes sur la plupart des modèles de cartes Java. De plus, certaines parties sont implémentées en natif ce qui permet d'obtenir de bonnes performances.

6.2.1 La solution GlobalPlatform

Comme nous l'avons expliqué dans le chapitre 3, les spécifications GlobalPlatform proposent deux méthodes pour établir des canaux sécurisés. La première est basée sur l'utilisation de clés symétriques, la seconde repose sur des clés asymétriques et l'utilisation de certificats numériques. Dans cette section nous étudions l'exploitation de chacune de ces deux méthodes dans le contexte de la grille de cartes à puce.

6.2.1.1 Le modèle symétrique

Dans le modèle symétrique, deux protocoles sont définis : *SCP01* et *SCP02* [65]. La première étape pour l'établissement d'un canal sécurisé, selon *SCP01-02*, est une phase d'authentification mutuelle durant laquelle la carte et l'utilisateur vérifient l'identité de leur correspondant. En générant des cryptogrammes, construits à partir de challenges (nombres aléatoires) qu'elles échangent en clair , les deux entités doivent prouver qu'elles partagent un secret initial (un jeu de clés symétriques, dites clés statiques[1]), et donc qu'elles peuvent se faire confiance. À l'issue de cette phase d'authentification, les deux parties disposent finalement d'un jeu de clés symétriques appelées clés de session et qui sont des dérivées des clés statiques. Ces clés de session sont employées pour assurer la confidentialité des données et l'intégrité des messages dans le reste de la communication.

Ce modèle de sécurité fonctionne très bien dans le mode classique d'utilisation de la carte à puce où l'on a un accès physique direct à la carte. Cette configuration permet à l'utilisateur d'être certain qu'il s'agit bien d'une carte, et plus précisément de sa propre carte. Ceci n'est malheureusement plus vérifié dans le cadre d'un système distribué où plusieurs utilisateurs peuvent accéder à distante à la carte à travers un réseau ouvert. En effet, du fait que les clés statiques (de type symétriques) sont connues par tous, un utilisateur mal intentionné pourrait sans peine se faire passer pour une carte. Les procédures de construction de ces clés sont publiées dans les spécifications GlobalPlatform est sont donc connues par tous. Les challenges étant échangés en clair, les clés de session peuvent donc être facilement régénérées par n'importe quel agresseur connaissant les clés statiques, et qui a pu observer la communication entre la carte et l'utilisateur depuis son début. Une première solution consiste à attribuer des clés statiques propres à chaque utilisateur de la grille. Toutes les clés statiques définies doivent alors être embarquées sur chacune des cartes. Toutefois, vues les limitations de mémoire des cartes, il n'est pas possible d'y charger un nombre trop important de clés ce qui limiterait donc le nombre des utilisateurs potentiels.

Nous en concluons que la méthode symétrique pour la grille de cartes à puce présente quelques inconvénients. Comme nous venons de l'expliquer ci-dessus, dans un contexte distribué multi-utilisateur, tel que la plate-forme grille de cartes à puce, où les communications se font à travers un réseau ouvert, plusieurs types d'attaques peuvent être facilement menés. Disposant des clés symétriques connues de tous, un agresseur peut par exemple déchiffrer les messages APDU, voire les modifier. Il pourra également se faire passer pour une entité authentique et usurper l'identité de l'un des deux participants sans que l'autre protagoniste ne puisse s'en apercevoir.

[1] Les clés statiques seront ensuite dérivées en clés de session, de type symétriques également, qui seront utilisées pour assurer la confidentialité et l'intégrité des messages échangés via le canal sécurisé.

6.2.1.2 Le modèle asymétrique

Depuis sa version 2.2 [66], GlobalPlatform a introduit un troisième protocole, *SCP10*, qui utilise la cryptographie asymétrique. Malheureusement, les cartes qui sont à notre disposition ne sont pas aussi récentes et implémentent uniquement les spécifications GlobalPlatform 2.0.1 [64] (précédemment appelées OpenPlatform), dans lesquelles seul le protocole *SCP01* est supporté. De plus, lorsque nous avons étudié la problématique de la sécurisation des communications pour la grille de cartes à puce, seule l'authentification avec des clés symétriques était définie ; la deuxième méthode d'authentification utilisant les clés asymétriques n'était pas encore publiée à l'époque. De plus, avec cette approche asymétrique, les problèmes d'établissement de canaux sécurisés dans le contexte de la grille ne sont pas tous résolus. Une phase d'identification est introduite au début de la communication. L'identification est réalisée grâce à des certificats numériques. La carte doit vérifier le certificat de l'entité extérieure afin de s'assurer de sa validité. Le principal inconvénient de cette méthode est que le processus de vérification s'effectue à l'intérieur de la carte. Étant données les ressources limitées des cartes à puce, cette manière de procéder n'est pas très efficace du point de vu performance. En outre, l'absence au niveau de la carte de certaines fonctionnalités spécifiques pour le traitement des certificats (telles qu'une horloge interne, une implémentation des formats d'encodage de certificats, etc.) constitue un vrai obstacle à la réalisation du processus de validation : il est possible que la carte ne parvienne pas à effectuer la vérification de la validité de certificats (péremption par exemple). Ce point sera détaillé dans la section 6.3.1.

6.2.2 Vers une solution générique

Afin de sécuriser les communications entre la carte et son utilisateur, nous proposons une solution qui consiste à établir des canaux sécurisés entre les deux parties. On veut assurer les propriétés suivantes :

- identification du correspondant : la carte et l'utilisateur doivent pouvoir identifier leur homologue. L'utilisation de certificats est aujourd'hui l'une des solutions les plus courantes pour résoudre des problèmes de sécurité liés principalement à l'identification et à l'authentification. Citons à titre d'exemple la méthode d'identification et d'authentification pour les réseaux sans fil [125] mise en œuvre par le protocole EAP [124] (*Extensible Authentication Protocol*). Pour ce qui nous concerne, nous attribuons un certificat à chacun des deux protagonistes (la carte et l'utilisateur). En début de communication, les deux entités échangent leurs certificats et peuvent ainsi vérifier l'identité de leur interlocuteur (via une entité extérieure).
- vérification des droits d'accès : la carte doit vérifier que l'utilisateur possède les droits d'accès lui permettant d'exécuter une application donnée. À cette fin, nous avons défini une politique centralisée pour la gestion des droits, basée sur une autorité de contrôle d'accès, que nous présentons dans la section 6.5.
- authentification : l'authentification consiste à s'assurer de la véracité de l'identité annoncée par l'autre partie. Ceci est réalisé grâce à un mécanisme de type challenge/réponse. Il s'agit pratiquement de l'authentification mutuelle définie dans le protocole *SCP01* (rappelons que nous ne disposons que de cartes compatibles *SCP01*) à laquelle nous avons apporté quelques modifications mineures, comme nous le présenterons dans le section 6.6. Ce modèle d'authentification reste valable pour le protocole *SCP02*, et dans l'hypothèse où nous aurions des cartes compatibles avec les spécifications GlobalPlatform 2.2 (c'est-à-dire supportant *SCP10*), d'autres adaptations seraient encore très probablement nécessaires, du fait des problèmes de vérification de certificat à l'intérieur de la carte comme nous allons l'expliquer dans la section 6.3.1.
- confidentialité et intégrité : pour assurer ces propriétés, nous utilisons les mêmes procédures et fonctions que celles employées dans GlobalPlatform.

La vérification de certificat constitue le point le plus critique pour la carte à puce. La réalisation de cette tâche repose sur l'utilisation d'un certain nombre de fonctionnalités spécifiques, notamment une horloge interne et une implémentation des normes et des formats d'encodage de certificats, qui ne sont malheureusement pas disponibles sur les cartes à puce. Pour remédier à ces insuffisances, nous avons défini un protocole de validation en ligne des certificats. Il s'agit d'un protocole dédié pour

les matériels à ressources limitées et en particulier pour les cartes à puce. Nous le décrivons dans la section 6.4.

6.3 Besoin de vérification par délégation

6.3.1 Quelques problèmes pour la vérification des certificats par la carte

Au vu des limitations matérielles et logicielles décrites ci-dessous, la vérification de la validité de certificats à l'intérieur de la carte ne semble pas être la meilleure solution. Le peu de mémoire disponible, la faible capacité de calcul et l'absence d'une API dédiée rendent le processus extrêmement difficile et inefficace. Il est probable que la carte ne puisse pas réaliser elle-même cette vérification, ou qu'elle n'arrive pas à conclure. Parmi les problèmes de la vérification de certificats à l'intérieur de la carte, on peut retenir :

- la taille du buffer d'entrée/sortie de la carte qui est, dans le meilleur des cas, limitée à 255 octets. Or, un certificat fait généralement entre 640 et 1600 octets. Par conséquent, il faut envoyer plusieurs commandes APDU afin que la carte puisse récupérer l'intégralité du certificat et commencer la phase de vérification. Après chaque commande transmise, il est nécessaire d'attendre la réponse retournée par la carte pour pouvoir lui envoyer la commande suivante. De plus, la vitesse de communication très lente (entre 9600 et 115200 bauds) va entraîner un coût important en temps pour l'envoi d'un certificat.
- la définition des certificats numériques utilise des normes et des notations bien déterminées. Par exemple, le format de certificat X.509 [123], qui représente le type de certificat le plus répandu, est défini par la norme ASN.1 [44]. ASN.1 est un langage informatique complexe et puissant qui permet de décrire et de spécifier des structures de données. Afin de pouvoir échanger des données ASN.1, des règles de codage ont été définies permettant de décrire la représentation de bits (bit le plus fort ou le plus faible d'abord, bits de parité, etc). Des types de codage tel que BER ou DER [44] sont utilisées afin de traduire des données ASN.1 en octets afin qu'elles puissent être traitées au niveau application. Une implémentation d'un compilateur ASN.1 (autrement dit d'un décodeur) permettant d'appliquer les règles de codage devrait donc être disponible sur la carte afin de pouvoir encoder/décoder les certificats. Ce n'est pas le cas aujourd'hui.
  Toutefois la partie 15 de la norme universelle pour les cartes à puce, l'ISO-7816[2] définit une syntaxe commune (conforme à la la norme ASN.1) pour la représentation d'informations cryptographiques, en particulier les certificats. Ce format est utile principalement pour la sauvegarde et l'échange de données cryptographiques mais ne représente pas une solution pour les problèmes liés à la vérification de certificats.
- la carte ne dispose pas d'une alimentation électrique propre lui permettant de maintenir une horloge interne. Il lui est donc impossible de vérifier si la date de fin de validité d'un certificat est atteinte ou pas.
- la construction et la vérification d'un chemin de certificats, lorsqu'il en existe un, sont coûteuses en temps de traitement.

Par ailleurs, le processus de vérification d'un certificat ne se limite pas à vérifier sa signature, sa date d'expiration, etc., mais consiste aussi à s'assurer qu'il n'a pas été révoqué. En effet un certificat peut être révoqué avant son expiration pour diverses raisons : les informations concernant le propriétaire ont changé (nom, rôle, etc.), sa clé privée a été compromise, etc. Il existe aujourd'hui deux façons principales de vérifier l'état d'un certificat. La première consiste à consulter une liste de révocation [123] au niveau de l'autorité de dépôt de l'architecture PKI. La deuxième façon consiste à utiliser un protocole de validation en ligne tel que OSCP [122] ou SCVP [43].

[2]Il ne faut cependant pas être très optimiste. Il existe aujourd'hui certains aspects qui sont normalisés mais non implémentés. À titre d'exemple, la partie 7 de la norme ISO 7816 [84] (intitulé *Interindustry commands for Structured Card Query Language (SCQL)*) définie depuis 1999 n'a pas été intégrée sur les cartes à puce, au moins pour ce qui concerne la plupart des Java Cards. La structure des données pour l'organisation de fichiers, définie dans la même norme ISO, n'est pas non plus implementée sur les cartes Java.

Pour conclure, nous pensons qu'il serait bénéfique et avantageux que la vérification des certificats s'effectue en dehors de la carte. C'est pourquoi la solution que nous proposons est basée sur la délégation : la carte confie la vérification des certificats à une entité extérieure, qui doit être une entité de confiance. En plus des vérifications syntaxiques et cryptographiques du certificat, cette entité utilise une des méthodes classiques (CRL, SCVP, OCSP, ou autre) afin de vérifier son état de révocation.

6.3.2 Vérification par délégation

Grâce à la délégation, l'utilisateur du certificat (la carte dans notre cas) donne procuration à une partie tierce (l'entité déléguée) pour que cette dernière effectue la vérification du certificat à sa place. Cette façon de procéder permet d'éviter des dégradations de performances, mais surtout d'éviter que la carte ne fournisse une réponse non conclusive (lorsque par exemple le certificat a expiré ou a été révoqué) ce qui pourrait arriver si la vérification se faisait à l'intérieur de la carte. La procédure se déroule comme suit. Sur demande de son client (la carte), l'entité déléguée vérifie la validité d'un certificat donné. Elle renvoie ensuite une réponse à la carte en précisant le résultat de la vérification (valide, non valide, ou autre). Un problème d'authentification se pose à ce niveau : comment la carte peut-elle être sûre que c'est bien l'interlocuteur valable (c'est-à-dire l'entité déléguée) qui répond à ses requêtes, et non pas une entité non autorisée ?

Afin de résoudre ce problème, nous attribuons à l'entité déléguée un certificat d'identité. Cette entité utilise sa clé privée pour signer les messages qu'elle envoie. Ainsi, et en se servant de la clé publique contenue dans ce certificat, la carte peut authentifier les données provenant de l'entité déléguée et vérifier leur intégrité. Il est indispensable que le certificat soit installé au préalable sur la carte.

En utilisant la délégation, nous proposons à la section 6.4 une solution pour la vérification en ligne de la validité de certificats[3], conçue spécialement pour les cartes à puce.

6.4 Notre contribution : le protocole CVPforScard

Certificate Validation Protocol for Smart Card ou tout simplement CVPforScard est un protocole de validation en ligne de certificats dédié aux matériels à ressources limitées, et en particulier aux cartes à puce.

6.4.1 Architecture générale de CVPforScard

La mise en œuvre de CVPforScard fait intervenir un certain nombre d'acteurs représentés sur la figure 6.1. Nous distinguons :

- l'application cliente ou application extérieure : c'est l'application côté utilisateur qui présente à la carte le certificat de l'utilisateur afin d'établir un canal sécurisé avec elle.
- l'application carte : il s'agit d'une application se trouvant sur la carte et qui doit vérifier les certificats qui lui sont envoyés. Cette fonctionnalité constitue un service global utilisable par l'ensemble des applets embarquées. Pour cela, nous avons défini une application dédiée, l'agent de validation (voir item suivant), permettant de réaliser cette fonction.
- L'agent de validation (ou de vérification) embarqué : en appelant les services de l'agent de validation embarqué, une applet pourra déclencher la procédure de vérification d'un certificat donné afin de s'assurer de sa validité. L'agent de vérification réalise les fonctions nécessaires au

[3]Une solution comparable à la notre a été présentée dans la littérature [114]. Dans la solution décrite, les requêtes de vérification envoyées par la carte sont traitées par la machine à laquelle elle est directement liée. Dans la solution que nous proposons, ces requêtes sont traitées par une entité de confiance à travers un canal sécurisé assurant l'intégrité et le non-rejeu des messages. De plus, dans [114], le fichier certificat de l'utilisateur doit être envoyé en intégralité à la carte, alors que dans notre solution nous utilisons un chemin vers le certificat. L'entité de confiance va donc chercher le certificat depuis son chemin ce qui conduit à de meilleures performances. En effet, un certificat comporte généralement environ mille octets, ce qui nécessite plusieurs commandes/réponses APDU afin de pouvoir l'envoyer en totalité à la carte avant de pouvoir commencer la vérification. Dans la solution que nous proposons, nous nous contentons d'envoyer un chemin vers le certificat (par exemple une URL). Cette information peut être envoyée à la carte dans une seule commande, et la carte la transmet à son tour vers l'entité de confiance. Celle-ci va par la suite récupérer directement le fichier certificat depuis le chemin indiqué et éviter donc les messages APDU ce qui permet de gagner en performance.

déroulement du protocole CVPforScard côté carte. C'est le médiateur entre l'application carte et l'entité déléguée.
Pour une carte intégrant les spécifications GlobalPlatform, le processus de vérification de certificats pourrait être confié au domaine de sécurité puisque son rôle est de fournir aux applications un ensemble de services cryptographiques et sécuritaires.

- l'entité déléguée : elle est chargée d'effectuer les vérifications nécessaires (syntaxiques, cryptographiques, etc.) sur un certificat afin de s'assurer de sa validité. L'entité déléguée doit en particulier vérifier l'état de révocation du certificat. Pour cela, elle est en liaison avec un ou plusieurs serveurs permettant de réaliser cette tâche et elle implémente l'approche correspondante (par exemple OCSP, SVCP, CRL, etc.).
- le serveur de vérification : toute entité capable de vérifier si un certificat donné est révoqué ou non peut jouer le rôle de serveur de vérification. Dans le protocole OCSP par exemple, cette entité est appelée le répondeur OCSP (*OCSP responder*). Il est également possible de consulter une CRL ou d'utiliser le protocole SCVP.

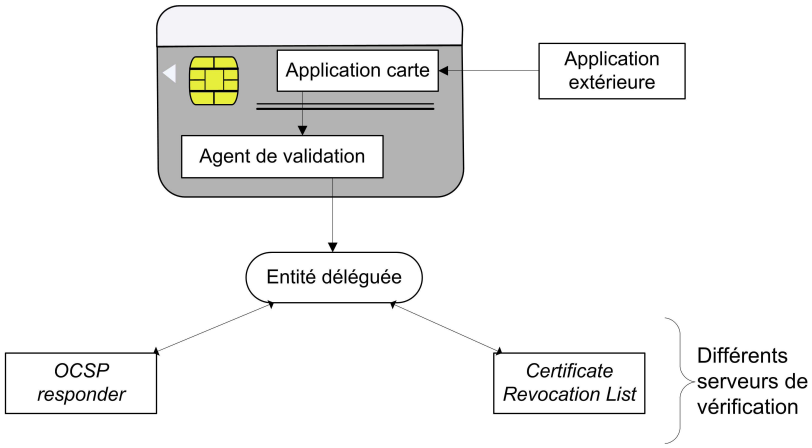


FIG. 6.1 – Schéma de vérification des certificats

Un problème important qui se pose pour la réalisation de cette solution est celui de la passivité de la carte à puce. Lorsqu'une carte reçoit une commande contenant le certificat de l'utilisateur de l'application extérieure, elle doit générer une requête de vérification de certificat et la transmettre à l'entité déléguée, et non pas à l'application extérieure qui avait envoyé la première commande. Une solution à ce problème, la proactivité simple, a été proposée et décrite en section 5.5.2.2. Cette solution permet à la carte d'encapsuler des requêtes dans les réponses APDU qu'elle envoie, afin de passer outre à sa passivité.

6.4.2 Introduction de nouveaux types de messages APDU

Afin de mettre en œuvre notre protocole, nous définissons un nouveau type de messages APDU : *Verify Certificate*. Ce type de message peut se présenter sous la forme de commandes ou de réponses. Nous en distinguons quatre variantes :

- la commande *Verify Certificate[path]* : cette commande permet à l'application extérieure d'envoyer à la carte un pointeur (par exemple une adresse HTTP) vers le certificat à vérifier.
- la réponse *Verify Certificate[request]* : il s'agit ici d'une réponse pro-active envoyée par la carte à l'entité déléguée. Les requêtes de vérification de certificats sont encapsulées dans des réponses *Verify Certificate[request]*.
- la commande *Verify Certificate[status]* : envoyée par l'entité déléguée, la commande *Verify Certificate[status]* lui permet de retourner à la carte le résultat de la vérification. Cette commande ne doit être envoyée que lorsque l'entité déléguée a déjà reçu une requête de type *Verify Certificate[request]*.

– la réponse *Verify Certificate[response]* : la carte se sert d'une réponse de ce type pour indiquer à l'application extérieure si le certificat présenté est valide ou non.

6.4.3 Fonctionnement du protocole

Dans la solution que nous proposons la vérification de certificat ne s'effectue pas à l'intérieur de la carte, il est donc inutile que la carte reçoive l'intégralité du certificat. En tant qu'utilisateur potentiel de certificat, la carte à puce n'aura besoin que des informations concernant la clé publique (c'est-à-dire la valeur de cette clé et les algorithmes avec lesquels elle doit être utilisée). En utilisant ces caractéristiques, nous limitons donc les communications avec la carte en terme de longueur de message et en terme de nombre de messages échangés, ceci afin d'obtenir de bonnes performances.

Nous supposons par ailleurs que l'utilisateur a pu récupérer le certificat de la carte, via l'entité de publication (le registre de configuration) en précisant le CID (*Card IDentifier*, identifiant unique par carte) de la carte en question, et qu'il a déjà effectué toutes les vérifications nécessaires (syntaxiques, cryptographiques, état de révocation, etc.) selon le modèle décrit dans la section 3.5.2 afin de s'assurer de sa validité. Cette fonction de vérification ne pose pas de problème pour l'utilisateur se connectant depuis une machine dotée de ressources matérielles et logicielles importantes et qui intègre donc toutes les fonctionnalités nécessaires pour la vérification des certificats. À la fin de cette étape, l'utilisateur dispose de la clé publique de la carte, clé qui sera utilisée par la suite dans la phase d'authentification mutuelle. La carte doit aussi vérifier la validité du certificat de l'utilisateur, ce que permet le protocole CVPforScard. Nous détaillons dans ce qui suit les étapes de déroulement de ce protocole illustré à la figure 6.2. Dans cette description, nous faisons abstraction de la gestion de la proactivité qui a été déjà présentée dans la section 5.5.2.2.

– Obtention de l'adresse du certificat : l'application extérieure commence par envoyer à la carte un pointeur vers le certificat utilisateur à vérifier. Le pointeur peut être fourni sous forme d'une adresse HTTP, FTP, d'une entrée dans un annuaire, d'une adresse de servlet, etc. Par la suite, l'entité déléguée doit être capable de télécharger l'intégralité de ce certificat depuis l'adresse indiquée. L'application extérieure utilise une commande de type de *Verify Certificate[path]* pour envoyer à la carte le pointeur vers le certificat. Le champ de données de cette commande contient le pointeur en question. Par exemple, s'il s'agit d'une adresse `http`, l'application extérieure doit envoyer une URL de la forme `http://www.server.fr/user/mycertificate.cer`.
– Formulation et envoi d'une requête de vérification : la carte, ou plus précisément son agent de validation, formule une demande de vérification. La construction d'une requête consiste principalement à envoyer à l'entité déléguée l'adresse depuis laquelle le certificat de l'utilisateur peut être récupéré. Cette requête est envoyée dans une réponse pro-active[4] de type *Verify Certificate[request]*. Une fois construite, la requête de vérification est envoyée vers l'entité déléguée[5].
– Vérification de la validité du certificat : grâce à l'adresse fournie par la carte, l'entité déléguée récupère le certificat de l'utilisateur et effectue les vérifications nécessaires. En plus des vérifications syntaxiques et cryptographiques, l'entité déléguée doit s'assurer que le certificat n'a pa été révoqué. Elle pourra à cet effet utiliser une approche de type CRL, SCVP, ou OCSP. Il s'agit du modèle de vérification que nous avons présenté dans la section 3.5.2.
– Renvoi du résultat : après vérification du certificat, l'entité déléguée retourne à la carte une réponse précisant si le certificat est valide ou non. Cette réponse est envoyée dans une commande de type *Verify Certificate[status]* et peut indiquer :
  • chemin invalide : l'entité déléguée n'arrive pas à récupérer le certificat depuis l'adresse qui lui a été fournie par la carte. L'adresse peut être mal formée, inexistante, ou non fonctionnelle.

[4]Cette réponse est en effet destinée à l'entité déléguée, et non pas à l'application extérieure, émettrice de la commande précédente. Comme nous l'avons expliqué dans la section 5.5.2.2, des valeurs particulières du champ SW permettent de faire la différence entre les réponses pro-actives et les réponses classiques.

[5]Il faut noter ici que la communication entre la carte et l'entité déléguée doit aussi être sécurisée. Nous cherchons principalement à assurer l'intégrité et le non rejeu des messages ainsi que l'authentification de leur origine (en particulier pour ce qui concerne l'entité déléguée). La carte doit être sûre que c'est bien l'entité déléguée qui lui fournit la réponse. Ce point sera détaillé dans la section 6.4.4.

- certificat invalide : si une des vérifications échoue, le certificat est considéré comme non valide. Plusieurs raisons peuvent en être la cause : certificat falsifié ou révoqué, validité expirée, chaîne de certification erronée, etc.
- vérification non concluante : lorsque le serveur de validation ne parvient pas à s'assurer de l'état de révocation du certificat, l'entité déléguée renvoie à la carte un message indiquant que ce dernier est dans un état inconnu.
- certificat valide : le processus de vérification a retourné un résultat positif quant à la validité du certificat. L'entité déléguée envoie donc à la carte une réponse pour indiquer que le certificat est bien valide.

Dans ce dernier cas, l'entité déléguée renvoie à la carte les informations concernant la clé publique contenue dans le certificat, à savoir, la valeur de cette clé et l'algorithme à utiliser. Si l'entité déléguée a besoin de plus d'une commande APDU pour envoyer toutes ces informations, elle utilise les champs P1 et P2 afin d'indiquer à la carte la présence de données supplémentaires. La carte renvoie alors une réponse particulière que nous avons définie (appelée *GetMoreData*) afin de récupérer la suite des données.

– Fin de la vérification : la carte renvoie une réponse à l'application extérieure lui indiquant si le certificat a été validé ou non.

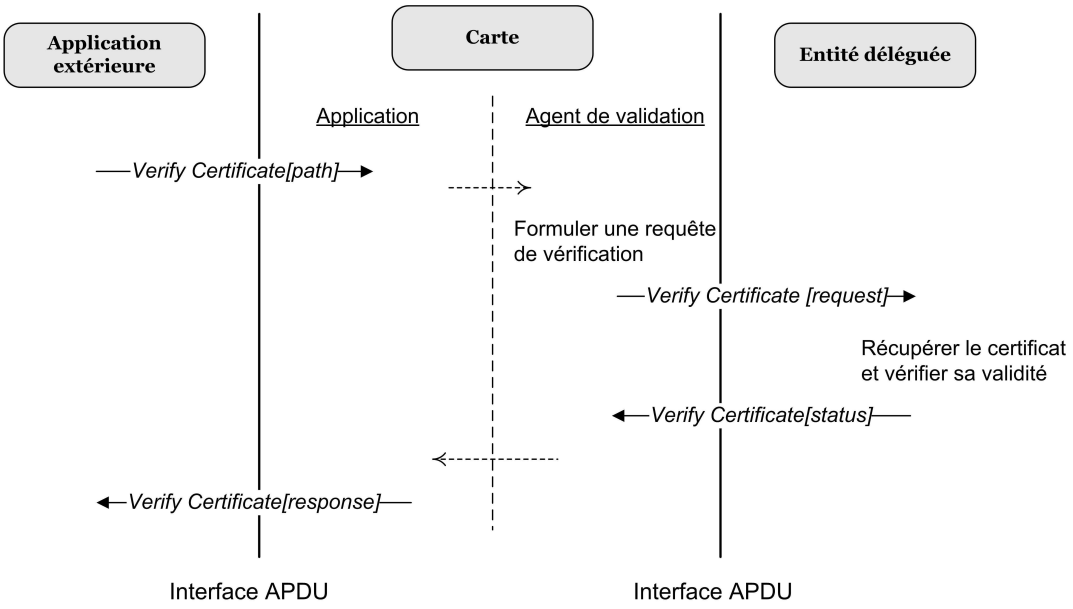


FIG. 6.2 – Schéma de vérification des certificats (abstraction faite de la gestion de la proactivité)

Le protocole CVPforScard présente l'avantage d'être indépendant de la méthode de vérification utilisée par l'entité déléguée, cette dernière recevant simplement un lien vers le fichier certificat. Elle peut donc implémenter n'importe quelle méthode de vérification (OCSP, CRL, ou autre). Cette propriété permet de garantir la compatibilité du protocole CVPforScard avec n'importe quelle approche. Ainsi, même si d'autres méthode de validation voient le jour, aucune modification de CVPforScard ne sera nécessaire.

6.4.4 Sécurisation de la communication entre la carte et l'entité déléguée

La sécurité de la solution proposée repose principalement sur l'authentification de l'entité déléguée. Les réponses de validation du certificat doivent donc être échangées à travers un canal sécurisé permettant à la carte de vérifier que c'est bien l'entité déléguée qui lui a envoyé la réponse, et non pas une entité non autorisée. Pour cela un certificat est attribué à l'entité déléguée ce qui permet à la carte de l'authentifier. Durant la communication, l'entité déléguée utilise sa clé privée pour signer les

messages à envoyer à la carte. Cette dernière, disposant de la clé publique associée, peut authentifier l'origine et vérifier l'intégrité des messages qu'elle reçoit. Par ailleurs, la carte doit s'assurer que les réponses reçues sont des réponses fraîches, et non pas des réponses à des requêtes antérieures. En effet, un intrus pourrait tenter de rejouer des réponses précédentes. Pour éviter ce type d'attaque, la carte génère un numéro qui permet d'identifier chaque couple <requête, réponse>. Le canal sécurisé établi entre la carte et l'entité déléguée permet donc d'assurer :

Le non-rejeu : les attaques par rejeu peuvent être contrées par l'utilisation d'un identifiant (numéro) qui permet de garantir que la réponse reçue correspond à la requête précédente, et n'est pas une copie d'une réponse antérieure. Afin d'éviter d'avoir des numéros identiques entre les différentes cartes, l'identifiant attribué à chaque requête doit être propre à une carte donnée. La construction de ce numéro est basée sur le CID, unique pour chaque carte, et plus précisément sur le code de hachage de celui-ci. L'unicité du CID permet de garantir l'unicité du condensé pour chaque carte (voir section 3.3.1.3). Un compteur de séquence est ajouté au code de hachage formant ainsi un identifiant unique pour chaque couple <*requête, réponse*>.

L'authentification de l'entité déléguée : afin de contrecarrer les attaques de type *man in the middle*, la carte doit authentifier l'origine des messages qu'elle reçoit. En effet, l'authentification permet d'éviter que la carte traite des réponses provenant d'une entité hostile. Chaque réponse renvoyée par l'entité déléguée est signée à l'aide de sa clé privée. Lorsque la carte reçoit une réponse à sa requête, elle vérifie l'intégrité et l'origine de la réponse qu'elle reçoit en s'assurant de l'authenticité de la signature. La carte utilise à cet effet la clé publique contenue dans le certificat de l'entité déléguée qui a été chargé au préalable.

Par ailleurs, les requêtes émanant de la carte peuvent être transmises en clair et ne nécessitent pas une forme de sécurisation particulière puisque la demande de vérification envoyée par la carte ne contient aucune donnée secrète ou sensible. Cependant, il est possible d'utiliser un canal sécurisé permettant d'assurer la confidentialité et/ou l'intégrité des données. Dans ce cas, la carte utilise sa propre clé privée pour signer la requête et la chiffre à l'aide de la clé publique de l'entité déléguée. À la réception, l'entité déléguée peut déterminer la carte ayant envoyé la requête grâce au numéro de cette requête envoyé en clair et contenant le code de hachage de l'identifiant de la carte (CID). Disposant des différents CID et des codes de hachage correspondants, l'entité déléguée peut alors connaître la carte émettrice de la requête et utilise alors la clé publique de la carte concernée afin de vérifier l'intégrité du message.

6.5 Gestion des droits d'accès

Un autre aspect de sécurité aussi important que la vérification d'identité, est celui du contrôle des droits d'accès. Avant qu'un utilisateur puisse exécuter une application, la carte doit vérifier s'il en a le droit. Pour cela nous avons défini une autorité de contrôle responsable de la gestion des droits d'accès. L'autorité de contrôle assure donc deux fonctions de vérification. La première concerne les droits d'accès. La deuxième est liée à la validité du certificat de l'utilisateur. Elle joue ainsi le rôle de l'entité déléguée définie dans la section 6.3.2.

Pour ce qui est droits d'accès, nous attribuons à chaque utilisateur de la grille, identifié par son certificat, la liste des applications (sous forme d'une liste d'AID) auxquelles il a le droit d'accéder. Une fois qu'il a sélectionné une applet, l'utilisateur envoie à la carte une commande contenant son identité et une URL vers son certificat. La carte pourra alors interroger l'autorité de contrôle afin de vérifier si l'utilisateur possède les droits nécessaires. La requête envoyée par la carte à l'autorité de contrôle est constituée de l'AID de l'applet couramment sélectionnée et de l'URL vers le certificat fourni par l'utilisateur. L'autorité de contrôle récupère le certificat depuis l'adresse indiquée et vérifie si son détenteur possède le droit d'accès à l'application dont l'AID est indiqué dans la requête de la carte. Peu importe l'ordre des opérations de vérification du certificat et des droits d'accès (pouvant dépendre éventuellement de la configuration de la machine et de l'implémentation du protocole), l'entité déléguée pourrait commencer par l'une ou l'autre. Dans le cas où l'entité déléguée commence par vérifier les droits d'accès, deux situations peuvent se présenter :

a) si l'utilisateur possède le droit d'exécuter l'application concernée, l'autorité de contrôle procède à la vérification du certificat et renvoie à la carte le résultat de cette vérification (voir figure 6.3) ;

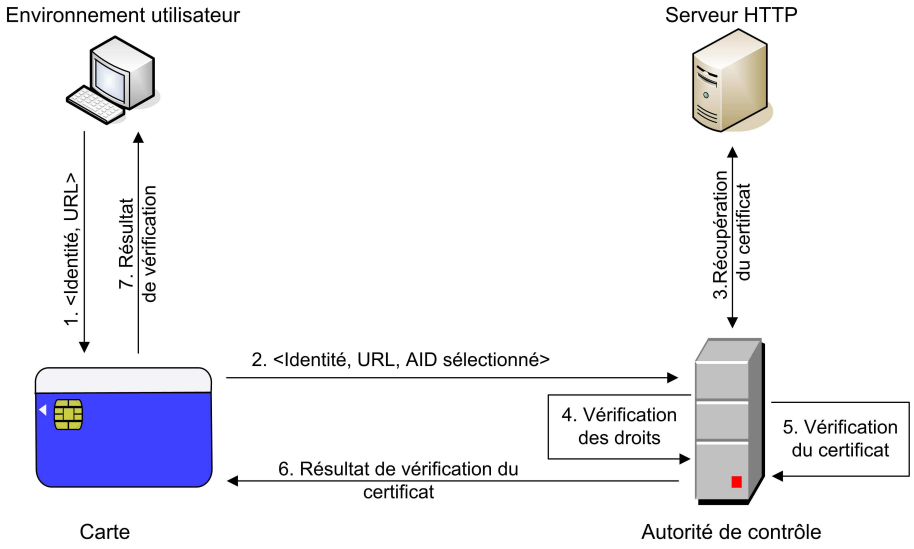


FIG. 6.3 – Vérification de droits d'accès : cas positif

b) sinon, elle renvoie à la carte une réponse indiquant "permission non accordée" (voir figure 6.4).

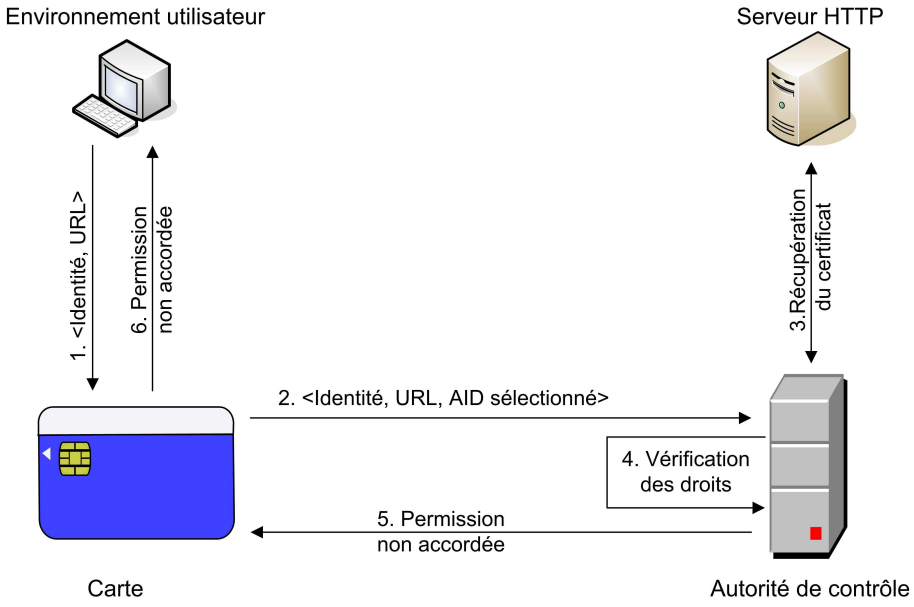


FIG. 6.4 – Vérification de droits d'accès : cas négatif

6.6 Authentification et établissement du canal sécurisé

Grâce aux étapes précédentes, les deux parties ont pu établir la validité du certificat de leur interlocuteur et l'identifier grâce aux informations associées. À l'issue de cette phase chacun des deux participants dispose de la clé publique de l'autre entité. Malheureusement, ceci ne garantit rien quant à l'identité réel des interlocuteurs : n'importe qui pourra récupérer le certificat d'une autre entité

et se faire passer pour elle (usurpation d'identité). Il est donc indispensable d'introduire une phase d'authentification mutuelle durant laquelle chaque participant doit prouver l'identité qu'il a annoncée. Nous utilisons pour cela les spécifications GlobalPlatform. La solution que nous proposons s'appuie sur le modèle d'authentification mutuelle du protocole *SCP01* que nous avons décrit dans la section 3.6.2. Ses principales étapes, représentées à la figure 6.5, sont les suivantes :

- l'entité extérieure génère et envoie à la carte un *host Challenge* (nombre aléatoire).
- la carte génère à son tour un *Card Challenge*. En utilisant les challenges (*Card Challenge* et *Host Challenge*) et les clés statiques partagées, la carte génère :
  - des clés de session (de type symétriques), utilisées pour sécuriser les messages dans le reste de la communication,
  - un cryptogramme appelée *Card Cryptogram*.

  Le *Card Challenge* et le *Card Cryptogram* sont ensuite retournés vers l'entité extérieure.
- l'entité extérieure, disposant de toutes les données initiales, produit à son tour :
  - les clés de session,
  - le même *Card Cryptogram*, et en effectuant une comparaison avec la valeur reçue peut authentifier la carte,
  - un *Host Cryptogram*, qui est renvoyé à la carte.
- la carte génère le même *Host Cryptogram* afin d'authentifier l'entité extérieure et envoie une réponse pour confirmer la phase d'authentification.

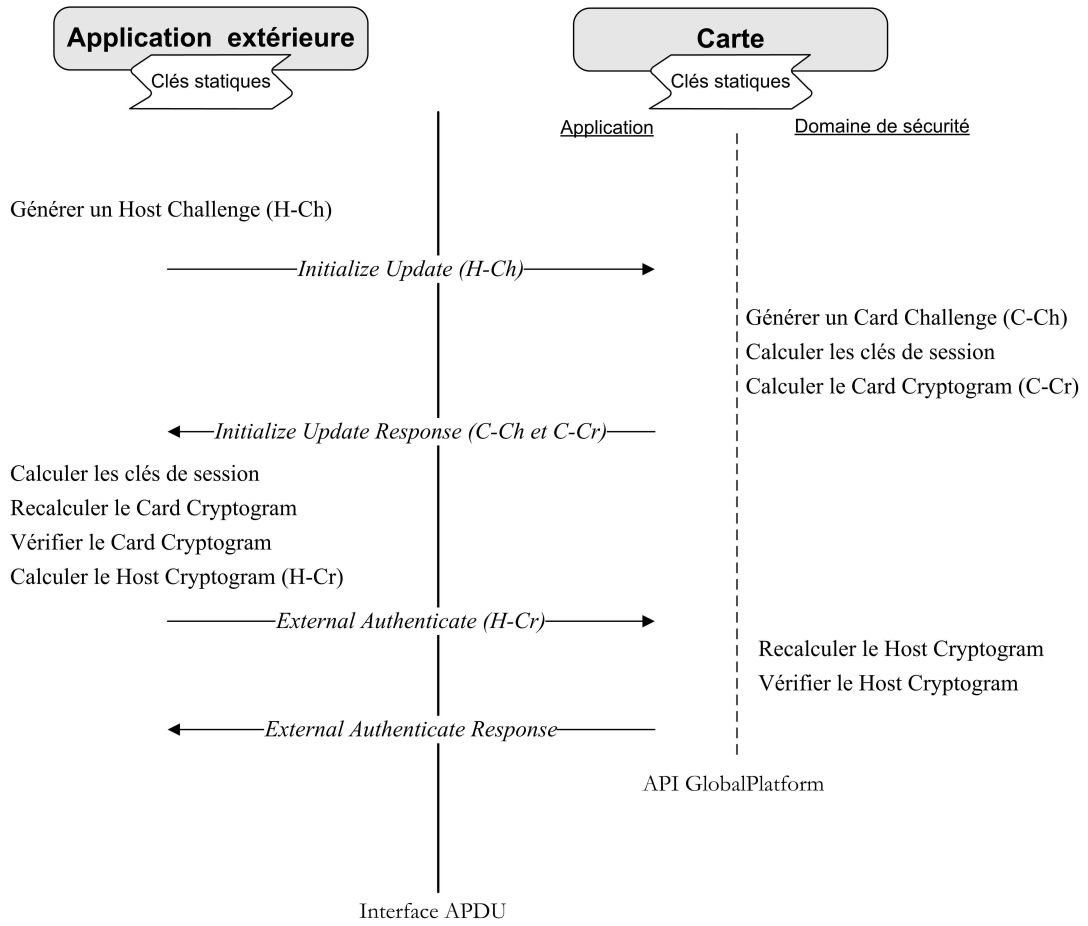


Fig. 6.5 – Schéma d'authentification mutuelle selon *SCP01*

Cependant, comme nous l'avons expliqué au début de ce chapitre (section 6.2.1.1), le modèle symétrique présente quelques problèmes pour un contexte distribué multi-utilisateur. L'intégrité des messages et la confidentialité des données sont mises en cause du fait que les clés de session, utilisées

pour assurer ces deux propriétés, peuvent être facilement reconstruites. En effet, la fabrication de ces clés est basée sur des clés symétriques connues par tous (les clés statiques), et des données (les challenges) échangées en clair ; les fonctions et les procédures de génération de ces clés sont décrites quant à elles dans les spécifications publiées par GlobalPlatform. Les attaques de type usurpation d'identité ne sont pas bien difficiles à réaliser non plus. Un utilisateur mal intentionné pourrait facilement générer un *Card Cryptogram* (ou un *Host Cryptogram*) valide et se faire passer pour une entité authentique. Ces problèmes sont principalement liés au fait que l'on ne dispose pas d'un accès physique à la carte, contrairement à une utilisation classique dans laquelle l'utilisateur peut être sûr qu'il communique avec sa propre carte.

Afin de résoudre ces problèmes, nous proposons une solution dans laquelle nous utilisons les paires de clés, privée et publique, des deux parties afin de générer les clés de session et les valeurs de cryptogrammes qui servent à l'authentification. Rappelons que chacun des deux participants dispose de la clé publique de son interlocuteur. Chaque protagoniste chiffre son propre challenge avec la clé publique de son correspondant. Seuls les détenteurs des clés privées correspondantes, c'est-à-dire la carte et le vrai utilisateur, peuvent déchiffrer les challenges et connaître leurs valeurs. Les clés de session générées à partir de ces challenges sont alors gardées secrètes et ne peuvent pas être régénérées par une autre entité. Il en est de même pour le *Card* et le *Host Cryptogram*. Ainsi, aucun intrus, même s'il dispose des clés statiques, ne pourra générer un cryptogramme valide. La figure 6.6 illustre cette nouvelle procédure.

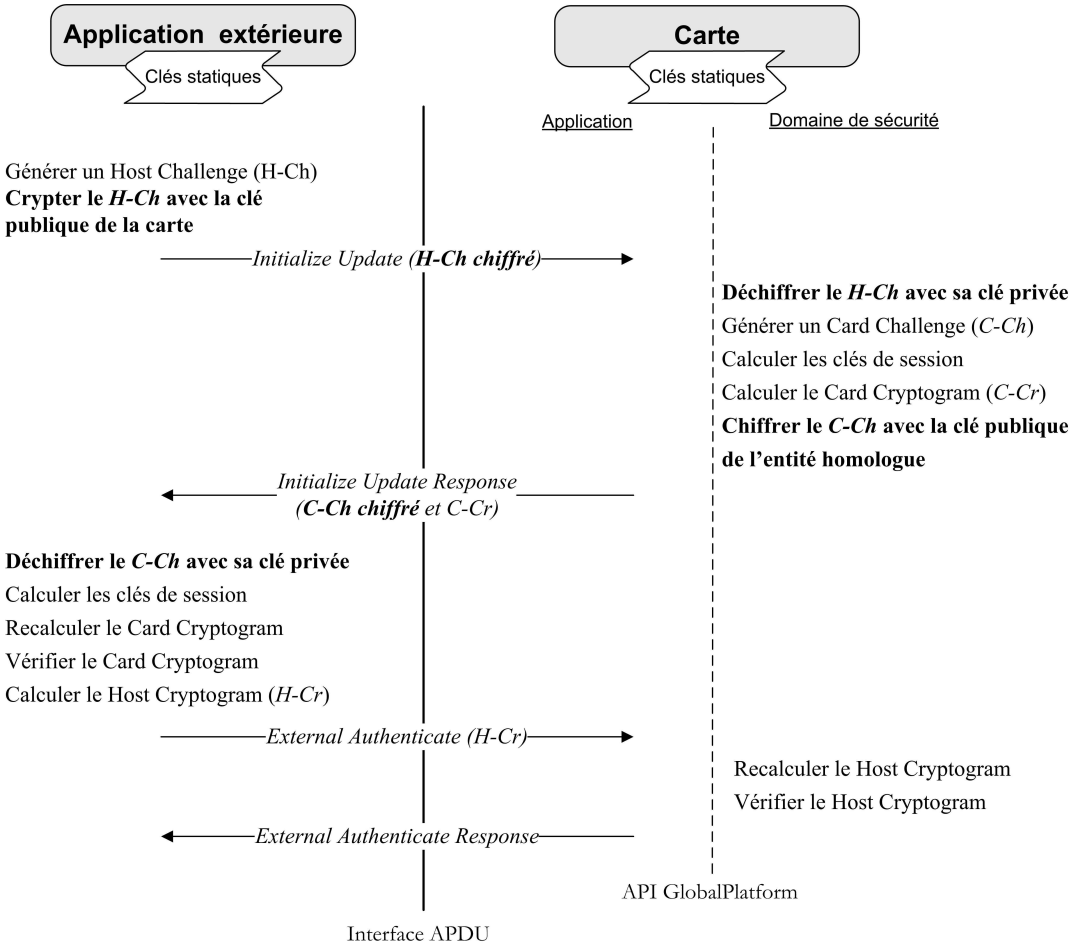


FIG. 6.6 – Nouvelle procédure pour l'authentification mutuelle

Les clés de session sont ensuite utilisées de façon classique pour assurer la confidentialité et l'intégrité des messages. La sécurité des réponses requiert aussi des clés de session. On peut utiliser les

mêmes clés ou en générer de nouvelles. Une méthode de génération consiste à appliquer l'algorithme DES sur une combinaison particulière des challenges, à la manière du protocole *SCP01*. Il s'agit du même principe de génération que celui utilisé pour les clés de session dans GlobalPlatform, ce qui garantit la qualité cryptographique de la solution proposée. Deux clés sont en fait nécessaires, une pour assurer la confidentialité des données et l'autre pour assurer l'intégrité des réponses. Comme dans le protocole *SCP02*, c'est l'algorithme DES qui est appliqué sur le champ de données de la réponse afin de garantir la confidentialité. Quant à l'intégrité, elle est réalisée par la génération d'un MAC. Afin d'assurer l'enchaînement des réponses et éviter les attaques de type rejeu, le MAC de la réponse courante doit être fonction du MAC précédent. Le MAC est généré en appliquant la fonction DES sur le champ de donnée de la réponse et de la commande correspondante (à la manière de *SCP02*). Une fois construit, le MAC généré est rajouté à la fin de la réponse à envoyer comme le montre la figure 6.7. Nous aboutissons enfin à un système parfaitement sécurisé dans lequel les commandes et les réponses APDU sont protégées.

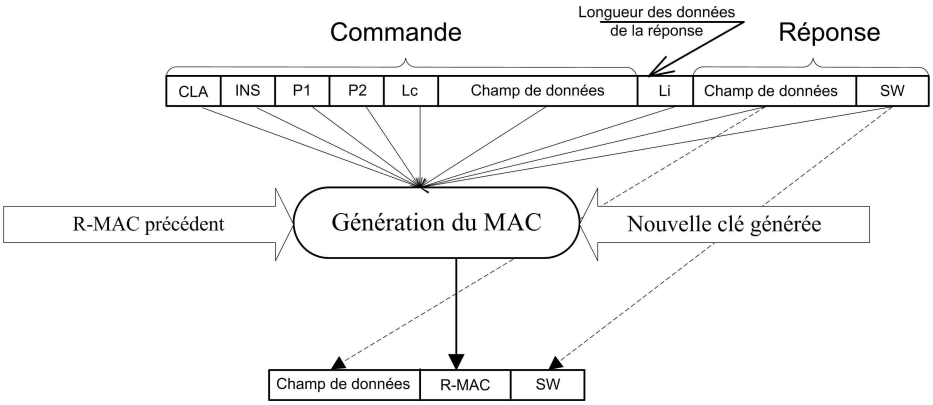


Fig. 6.7 – Génération du MAC pour les réponses APDU

6.7 Validation formelle du protocole de communication

La validation formelle de notre modèle de communication a été réalisée à l'aide de l'outil AVISPA [7] (*Automated Validation of Internet Security Protocols and Applications*). Cet outil est une plate-forme dédiée à la vérification et à la validation formelle des propriétés de sécurité (confidentialité, authentification, etc.) de protocoles de communication.

Dans AVISPA, les protocoles sont décrits en utilisant le langage *High Level Protocol Specification Language* (HLPSL) [36]. Ce langage permet de déclarer des objets de types basiques tels que des clés, des nonces, etc. Afin de permettre la spécification de propriétés de sécurité à vérifier, HLPSL fournit des primitives appelées prédicats tels que *secret* (pour la confidentialité), et *request* et *witness* (pour l'authentification).

L'architecture générale de la plate-forme AVISPA est présentée à la figure 6.8. Un convertisseur appelé *hlpsl2if* permet de traduire une description HLPSL en une représentation dans format intermédiaire (*Intermediate Format* : IF) composé d'états transitoires. La description HLPSL est réécrite selon un langage formel qui la traduit en un séquence d'instructions appelées états transitoires [137]. Le format IF peut être alors interpretable par les outils de bas niveau de l'environnement. AVISPA comprend en effet quatre outils de vérification qui sont :

OFMC (On-the-fly Model-Checker) [11]. OFMC effectue la vérification d'un protocole de façon itérative. Après traitement du format Intermédiaire (IF), l'outil parcourt les états transitoires spécifiés dans la représentation IF afin de chercher une éventuelle faille.

CL-AtSe (CL-based Attack Searcher) [143]. CL-AtSe utilise une approche basée sur les contraintes afin de détecter d'éventuelles failles de sécurité. Il emploie en outre des techniques heuristiques pour élimer les états redondants dans le but de simplifier le protocole.

SATMC (SAT-based Model-Checker) [4]. SATMC prend un état transitoire de IF (*Intermediate Format*) et cherche si cet état conduit à une violation de l'une des propriétés de sécurité.

TA4SP (Tree Automata-based Protocol Analyser) [18]. Pour vérifier le protocole décrit en entrée, TA4SP utilise des langages réguliers pour évaluer les possibilités qu'a l'intrus de violer les propriétés de sécurité du protocole (telles que l'intégrité, la confidentialité, etc.).

Ces différents outils analysent le protocole décrit en entrée afin de tester si les propriétés déclarées dans la section *goal* (voir section 6.7.1) du fichier HLPSL sont satisfaites ou au contraire violées. En cas de faille de sécurité, la trace des attaques correspondantes est générée.

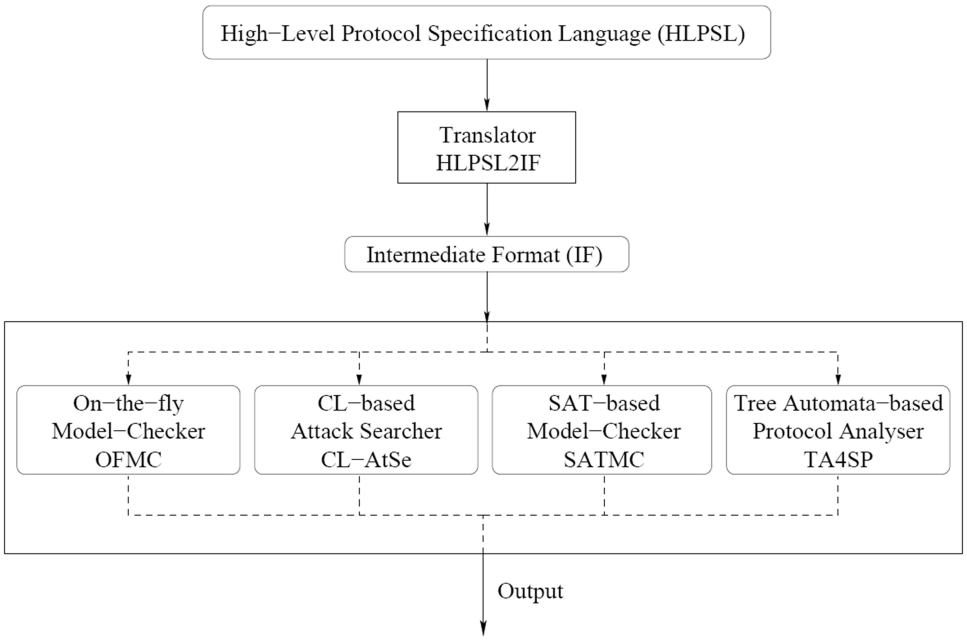


FIG. 6.8 – Architecture de AVISPA (source [138])

6.7.1 Structure générale d'un fichier HLPSL

Le langage HLPSL est basé sur la modélisation d'entités de type *rôle*. Un rôle définit des actions qui seront exécutées par un *agent* associé. Au début de la définition d'un rôle, on précise l'agent qui lui correspond par une ligne de commande de la forme (*played_by Agent*). Un rôle possède des paramètres d'entrée pouvant être par exemple des clés, d'autres agents qui interviennent dans le protocole, des canaux de communication (*channel (dy)*), etc., comme le montre le listing 6.1. Dans un rôle, il est possible de déclarer des variables locales, de type `text`, `symmetric_key`, etc.

La partie fondamentale dans un rôle est celle des transitions. À une transition est associé un ensemble d'actions. Celles-ci peuvent correspondre par exemple à la génération de valeurs intermédiaires, ou à l'envoi/réception de données. On notera que les variables qui portent le signe d'apostrophe (') dans la description HLPSL correspondent à des informations non connues avant la transition. Afin de vérifier des propriétés de sécurité (telles que la confidentialité et l'authentification) des prédicats peuvent aussi être associées à une transition. Comme exemples de prédicats, on peut citer *secret*, *request* ou *witness* que nous allons présenter dans la section 6.7.2.

La modélisation HLPSL d'un protocole comporte aussi deux rôles particuliers :

- le rôle *session* qui permet de décrire le déroulement global du protocole en effectuant la composition des différents rôles.
- le rôle *environment* qui permet d'introduire un agent particulier modélisant l'intrus pour lequel on testera s'il est capable de violer les propriétés de sécurité du protocole.

```
role alice(A, B, S : agent,
           Kas : symmetric_key,
           SND, RCV : channel (dy))
           %Canaux de communication

played_by A
def=
 local
    State: nat,
    Kab: symmetric_key,
    Na : text

    ...
end role
```

Listing 6.1 – Exemple de définition d'un rôle

```
role alice(...)
...

transition


    2. State = 2 /\ RCV({Nb'}_K) =|>
       State':= 3 /\ K1' := Hash(Na.Nb')
                  /\ SND({Nb'}_K1')
                  %Opération de chiffrement de la valeur Nb avec la clé K1
                 /\ witness(A, B, bob_alice_nb, Nb')

...
end role
```

Listing 6.2 – Exemple de transition

La dernière partie d'une spécification HLPSL est la section *goal* dans laquelle on précise les propriétés de sécurité à vérifier et qui correspondent aux prédicats définis dans les différents rôles du protocole.

6.7.2 Propriétés utilisées

Nous décrivons ici les deux propriétés fondamentales (confidentialité et authentification) que nous avons utilisées est dont la vérification est intégrée dans AVISPA.

La confidentialité

La propriété *secret* [137, 138], ou confidentialité, est modélisée au moyen d'un prédicat de type `secret(T, id, {A, B})` qui se traduit par "*la valeur de la variable T est un secret qui doit être partagé seulement entre les agents A et B*". La confidentialité de ce secret est violée chaque fois que l'intrus en apprend connaissance.

L'étiquette *id* (de type *protocol-id*) est employée pour identifier cette propriété de façon unique. On utilise cette étiquette dans la section *goal* pour indiquer que l'on souhaite vérifier cette propriété.

L'authentification

L'authentification est modélisée dans HLPLS par les prédicats *witness* et *request* employés selon la procédure suivante :

- lorsqu'une entité envoie un message devant être authentifié, elle utilise le prédicat `witness(A, B, na, Na)` qui se traduit par : "*A envoie un message à B, message devant être authentifié grâce à la donnée na*". L'étiquette Na (de type *protocol-id*) sert à identifier cette propriété de façon unique. Dans la section *goal* on utilise cette étiquette pour indiquer que l'on souhaite vérifier cette propriété.

– à la réception du message il faut utiliser le prédicat `request(B, A, na, Na)` qui se traduit par : "*l'agent B ayant reçu un message de A, a pu l'authentifier grâce à la données na*". Les deux étiquettes `Na` doivent être identiques dans les deux prédicats *witness* et *request* associées.

Le listing 6.3 présente un exemple d'utilisation des prédicats *secret*, *request* et *witness*.

```
role alice(...)
...

transition

     2. State = 2 /\ RCV({Nb'}_K) =|>
        State':= 3 /\ K1' := Hash(Na.Nb')
                   /\ SND({Nb'}_K1')
                   /\ secret(K1',k1,{A,B})
                   /\ witness(A ,B, bob_alice_nb, Nb') -----
                                                          |
...                                                       |
                                                          |
end role                                                  |
                                                          |
role bob(...)                                             |
                                                          |
transition                                                |
                                                          |
    2. State = 3 /\ RCV({Nb}_K1) =|>                      |
                                                          |
        State':= 4 /\ request(B, A, bob_alice_nb, Nb) -----

end role

...

goal
    secrecy_of k1
    authentication_on bob_alice_nb
end goal
```

Listing 6.3 – Modélisation des propriétés de confidentialité et d'authentification dans AVISPA

6.7.3 Modélisation de la solution proposée

Pour décrire notre protocole CVPforScard en HLPSL, nous avons défini un certain nombre d'agents qui correspondent aux principaux intervenants du système.

De façon classique, on se place dans le cas où on ne considère que les réponses positives, c'est-à-dire les réponses dont la vérification aboutit à un succès. Les autres réponses provoquent l'arrêt du protocole et un message d'erreur est renvoyé à l'utilisateur. Comme nous l'avons déjà évoqué, nous supposons que l'utilisateur dispose de la clé publique de la carte avec laquelle il souhaite entamer une communication. Cette clé publique est contenue dans le certificat qui peut être récupéré depuis le registre de configuration. On suppose que l'utilisateur a pu effectuer toutes les vérifications nécessaires pour s'assurer de la validité du certificat (voir section 6.4.3). De plus, il partage avec la carte un jeu de clés statiques.

Pour modéliser notre protocole[6], nous avons utilisé des canaux de communication de type Dolev-Yao [138] (*intruder model*). Dans ce modèle, l'intrus possède un contrôle total du réseau, autrement dit des canaux de communication. Il peut donc intercepter tous les messages envoyés par les différents participants. Il est aussi capable de modifier les messages et d'envoyer des données en tentant d'usurper l'identité d'un autre participant légitime ou honnête. La figure 6.9 présente les différents canaux que nous avons définis. Chaque agent dispose d'un canal de réception et d'un canal d'émission pour tout autre agent avec lequel il doit communiquer.

Nous présentons dans ce qui suit les différents agents définis dans la description HLPSL de notre protocole :

[6]L'intégralité de la description du protocole (définie en langage HLPSL) est donnée en annexe C.

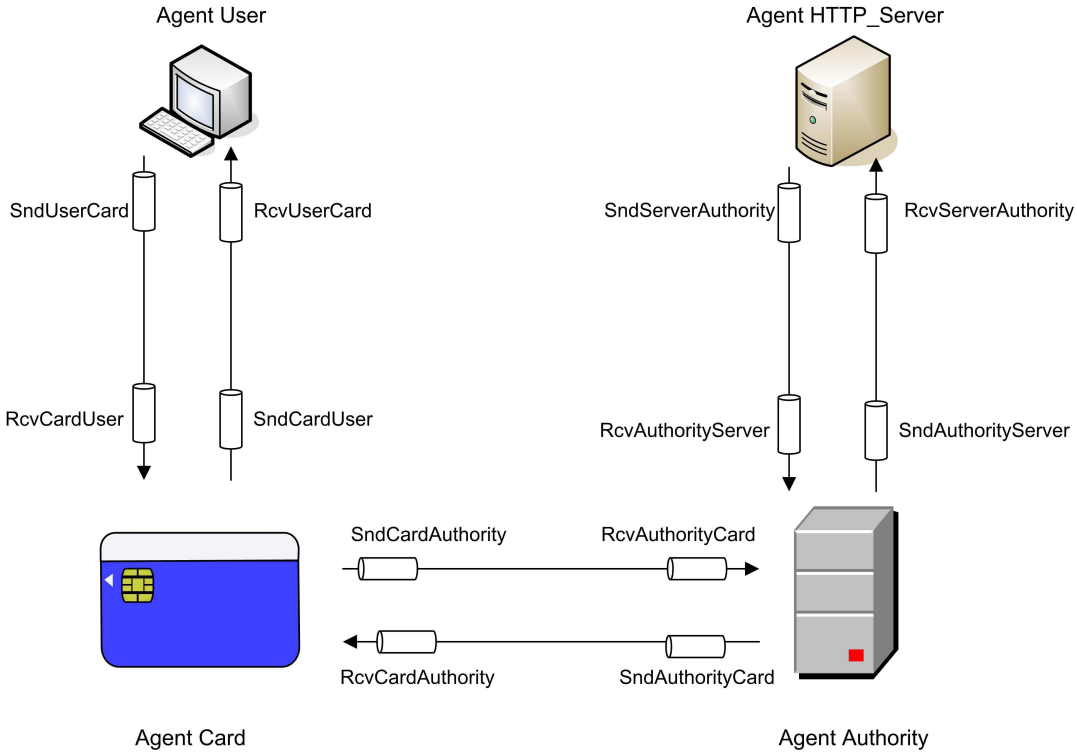


Fig. 6.9 – Les différents canaux de communication définis dans la modélisation de CVPforScard

– L'agent utilisateur (*User*). Il joue le rôle *cvp_Init* représenté sur le listing 6.4 et dont l'objectif est d'établir un canal sécurisé avec l'agent carte.
Ce rôle évolue de la façon suivante :
  • transition 1 : l'utilisateur envoie à la carte une commande de type *Verify Certificate[path]* dans laquelle il précise son identité et l'agent distant depuis lequel son certificat peut être téléchargé.
  • transition 2 : une fois qu'il a reçu de la carte le résultat de la vérification du certificat (que nous supposons *true*), l'utilisateur lui envoie une commande *Initialize Update* dans laquelle il chiffre son *Host Challenge*, avec la clé publique de la carte.
  • transition 3 : l'utilisateur reçoit sur son canal de communication avec la carte, une réponse dite *Initialize Update Response* contenant le *Card Cryptogram*. L'utilisateur génère alors des clés de session, qui doivent être gardées secrètes d'où l'utilisation du prédicat *secret(Session_Key', session_key2 , User, Card)*. De plus, il doit authentifier la carte grâce à son *Card Cryptogram* : *request (User, Card, card_cryptogram, Card_Cryptogram')*. À la fin de cette transition, l'utilisateur envoie à la carte un *Host Cryptogram* dans une commande *External Authenticate* ce qui permet à la carte de l'authentifier. Nous utilisons pour cela le prédicat *witness(User, Card, host_cryptogram, Host_Cryptogram')*.
  • transition 4 : l'utilisateur reçoit de la carte une réponse de type *External Authenticate Response* indiquant l'authentification de l'utilisateur et l'établissement d'un canal sécurisé.

– L'agent carte (*Card*). Dans le protocole SCVPforScard, la carte communique avec l'utilisateur et avec l'entité déléguée (agent *Authority*). Nous définissons pour cela quatre canaux de communication : `RcvCardUser` et `SndCardUser` qui correspondent respectivement au canal de réception et d'envoi avec l'agent *User* et `RcvCardAuthority` et `SndCardAuthority` pour le dialogue avec l'agent *Authority*. La carte est modéliseé par le rôle *cvp_Resp* présenté sur le listing 6.5.
Les transitions mises en œuvre sont les suivantes :

```
role cvp_Init (User, Card, HTTP_Server : agent, .........)

played_by User

  transition

  1. State = 0 /\ RcvUserCard(start) =|>
                  %|start est l'événement qui permet de déclencher le déroulement du protocole
                  %|autrement dit c'est le point d'entrée du protocole
alice to begin the protocol run
     State' := 1 /\ SndUserCard(User.HTTP_Server)


   2. State = 1 /\ RcvUserCard(true) =|>
      State' := 2 /\ Host_Challenge' := new()
                  /\ SndUserCard({Host_Challenge'}_(Pk_C))


   3. State = 2 /\ RcvUserCard({Card_Challenge'}_(Pk_U).
               {Card_Challenge'.Host_Challenge}_({Host_Challenge.Card_Challenge'}_(Static_Key))) =|>
                                          %|----------------Session Key-----------------|
               %|----------------------Card Cryptogram----------------------------------|

      State' := 3 /\ Session_Key':={Host_Challenge.Card_Challenge'}_(Static_Key)
                  /\ Card_Cryptogram':= {Card_Challenge'.Host_Challenge}_(Session_Key')
                  /\ Host_Cryptogram':= {Host_Challenge.Card_Challenge'}_(Session_Key')
                  /\ secret(Session_Key',session_key2 ,{User, Card})
                  /\ request (User, Card, c_cr, Card_Cryptogram')
                  /\ SndUserCard(Host_Cryptogram')
                  /\ witness(User, Card, h_cr, Host_Cryptogram')

   4. State =3 /\ RcvUserCard(true)=|>
       State':=4
```

Listing 6.4 – Modélisation de l'agent *User*

```
role cvp_Resp (Card, User, Authority, HTTP_Server : agent,
               ....
                RcvCardUser, SndCardAuthority, RcvCardAuthority, SndCardUser : channel(dy))

played_by Card

  transition

  1. State = 0 /\ RcvCardUser(User.HTTP_Server)=|>
    State' := 1 /\ ReqNum':=new()
                  /\ SndCardAuthority(ReqNum'.User.HTTP_Server)


  2. State = 1 /\ RcvCardAuthority(ReqNum.Pk_U.{H(ReqNum.Pk_U)}_inv(Pk_A))=|>
    State' := 2 /\ Signature':={H(ReqNum.Pk_U)}_inv(Pk_A)
               /\ request(Card, Authority, signature, Signature')
               /\ SndCardUser(true)


  3. State = 2 /\ RcvCardUser({Host_Challenge'}_(Pk_C))=|>
    State' := 3 /\ Card_Challenge' := new()
               /\ Session_Key':={Host_Challenge'.Card_Challenge'}_(Static_Key)
               /\ Card_Cryptogram':= {Card_Challenge'.Host_Challenge'}_(Session_Key')
               /\ secret(Session_Key',session_key1, {User, Card})
               /\ SndCardUser({Card_Challenge'}_(Pk_U).Card_Cryptogram')
               /\ witness(Card,User,c_cr,Card_Cryptogram')

  4. State = 3 /\ RcvCardUser({Host_Challenge.Card_Challenge}_Session_Key) =|>
                           %|-------------Host Cryptogram--------------|
    State' := 4 /\ Host_Cryptogram':= {Host_Challenge.Card_Challenge}_(Session_Key)
               /\ request(Card, User,h_cr,Host_Cryptogram')
               /\ SndCardUser(true)
```

Listing 6.5 – L'agent *Card*

• transition 1 : la carte reçoit de l'utilisateur une commande de type *Verify Certificate[path]* qu'elle transmet à l'agent *Authority* dans un message de type *Verify Certificate[request]*.

• transition 2 : la carte reçoit sur son canal de communication avec l'agent *Authority* la réponse à sa demande de vérification contenant la clé publique de l'utilisateur[7]. Elle authentifie l'expéditeur grâce à la signature du message par sa clé privée : *request(Card, Authority, signature, Signature')*. Elle envoie ensuite à l'utilisateur le résultat de la vérification, c'est-à-dire un message *true*.

• transition 3 : la carte reçoit de la part de l'utilisateur une commande *Initialize Update* contenant le *Host Challenge* chiffré. À partir des challenges, elle génère les clés de session qui doivent être connues seulement des deux parties *User* et *Card* : *secret(Session_Key', session_key1, User, Card)*. Ensuite elle envoie à l'utilisateur son *Card Challenge* et son *Card Cryptogram* à travers lequel ce dernier va pouvoir l'authentifier : *witness(Card, User, card_cryptogram, Card_Cryptogram')*.

• transition 4 : la carte reçoit de l'utilisateur une commande de type *External Authenticate*, contenant le *Host Cryptogam*. Elle authentifie l'utilisateur grâce à cette valeur : *request(Card, User, host_cryptogram, Host_Cryptogram')* et lui envoie par la suite un message de validation.

– L'agent *Authority*. Il correspond à l'entité déléguée. Il a pour fonction de traiter les requêtes de vérification des certificats qu'il reçoit de la carte et est représenté par le rôle *cvp_Verify* présenté sur le listing 6.6.

```
role cvp_Verify (Authority, HTTP_Server, Card, User : agent,
                 .......
                  RcvAuthorityCard, SndAuthorityServer, RcvAuthorityServer, SndAuthorityCard : channel(dy))


played_by Authority

  transition

  1. State = 0 /\ RcvAuthorityCard(ReqNum'.User.HTTP_Server)=|>
    State' := 1 /\ SndAuthorityServer(User)


  2. State = 1 /\ RcvAuthorityServer(Pk_U.User.{Hash(Pk_U.User)}_inv(Pk_CA))=|>
                                  %|------Certificat Utilisateur--------|
    State' := 2 /\ Signature':={H(ReqNum.Pk_U)}_inv(Pk_A)
                /\ SndAuthorityCard(ReqNum.Pk_U.Signature')
                /\ witness(Authority, Card, signature, Signature')
```

Listing 6.6 – L'agent *Authority*

Il est composé de deux transitions qui sont :

• transition 1 : l'entité déléguée reçoit de la carte (sur son canal `RcvAuthorityCard`) une requête de type *Verify Certificate[request]*. En utilisant son canal `SndAuthorityServer`, elle envoie à l'agent HTTP_Server (qui représente un serveur HTTP par exemple) une requête pour demander le certificat utilisateur.

• transition 2 : après avoir reçu sur son canal RcvAuthorityServer le certificat de l'utilisateur (depuis le serveur HTTP), l'agent *authority* renvoie à la carte (en utilisant le canal SndAuthorityCard) un message de type *Verify Certificate[response]*, dans lequel il met la valeur de la clé publique de l'utilisateur. Le message envoyé doit être signé afin que la carte puisse authentifier la source : *witness(Authority, Card, signature, Signature')*.

– L'agent *HTTP_Server*. Cet agent correspond à un serveur HTTP contenant le certificat de l'utilisateur. Il est représenté par le rôle *getCertificate* présenté sur le listing 6.7. Cet agent a juste pour fonction de renvoyer à l'entité déléguée (l'agent *Authority*) le certificat de l'utilisateur

[7]Rappelons que nous ne considérons que les réponses positives, les autres provoquant l'arrêt du protocole.

dont l'identité figure dans le message qu'il reçoit.

```
role getCertificate(.....)

played_by HTTP_Server

def= local
       State : nat

init State := 0

  transition

  1. State = 0 /\ RcvServerAuthority(User)=|>
    State' := 1 /\ SndServerAuthority(Pk_U.User.{Hash(Pk_U.User)}_inv(Pk_CA))
                                      %|------Certificat Utilisateur--------|

end role
```

Listing 6.7 – L'agent *HTTP_Server*

L'étape suivante consiste à établir la composition des différents rôles décrits précédemment ce qui revient à définir le rôle session présenté sur le listing 6.8.

```
role session(User, Card, Authority, HTTP_Server : agent,
            Pk_U, Pk_C, Pk_A, Pk_CA : public_key,
            H, Hash : hash_func,
            Static_Key: symmetric_key)
def=

  local SUC, RUC, RCU, SCA, RCA, SCU, RAC, SAS, RAS, SAC, SSA, RSA : channel (dy)

  composition
          cvp_Init(User, Card, HTTP_Server, Pk_U, Pk_C, Static_Key, SUC, RUC)
       /\ cvp_Resp(Card, User, Authority, HTTP_Server, Pk_C, Pk_A, H, Static_Key, RCU, SCA, RCA, SCU)
       /\ cvp_Verify(Authority, HTTP_Server, Card, User, Pk_A, Pk_CA, H, Hash, RAC, SAS, RAS, SAC)
       /\ getCertificate(HTTP_Server, Authority, User, Card, Pk_CA, Pk_U, Hash, SSA, RSA)

end role
```

Listing 6.8 – Le rôle *session*

Enfin, on définit le rôle environnement comme on peut le voir sur le listing 6.9. Dans ce rôle, nous introduisons un nouvel agent, l'intrus (*intruder*), et nous indiquons les données et les informations qu'il connaît initialement : les différents agents du système, les clés publiques et les clés statiques partagées.

La dernière partie de la description HLPSL est la définition de la section *goal*, présentée sur le listing 6.10 dans laquelle il faut préciser les propriétés que l'on veut vérifier. Les objectifs sont liés aux différents prédicats *secret*, *request* et *witness*, déjà mentionnés lors de la définition des rôles.

6.7.4 Vérification du protocole

Les outils sur lesquels nous avons exécuté notre protocole ont donné un résultat "*safe*" à l'exception de l'outil TA4SP qui retourne le message suivant "*Some rules may be not fired so TA4SP does not do the verification.*". Ceci est expliqué par le fait que certaines règles générées par le convertisseur HLPSL2IF ne sont pas encore prises en compte par TA4SP et donc ne peuvent pas être interprétées. Pour les autres outils (c'est-à-dire OFMC, ATSE et SATMC) aucune faille n'a pu être détectée. À titre d'exemple nous présentons au listing 6.11 le résultat produit par l'outil OFMC.

```
role environment()
def=

  const u, c, a, http_server : agent,
        static_key : symmetric_key,
         h, hash2 : hash_func,
      pku, pkc, pka, pki, pkca : public_key

  intruder_knowledge = {u, c, a, static_key, h, hash2, http_server, pki, pku, pkc, pka, pkca, inv(pki)}

  composition

        session(u, c, a, http_server, pku, pkc, pka, pkca, h, hash2, static_key) /\
        session(u, i, i, http_server, pku, pki, pka, pkca, h, hash2, static_key) /\
    [...]
```

Listing 6.9 – Le rôle *environnement*

```
goal

%Vérification de la confidentialité des clés de session établies entre la carte et l'utilisateur
 secrecy_of session_key1, session_key2

%Authentification mutuelle:

    %1- L'utilisateur cherche à authentifier la carte grâce à son Card_Cryptogramm
 authentication_on card_cryptogram
    %2- La carte cherche à authentifier l'utilisateur grâce à son Host_Cryptogramm
 authentication_on host_cryptogram

%La carte cherche à authentifier l'entité déléguée grâce à sa signature
  authentication_on signature


end goal
```

Listing 6.10 – La section Goal

```
% OFMC
% Version of 2006/02/13
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/avispa/web-interface-computation/./tempdir/workfileA10131.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 3.17s
  searchTime: 18.02s
  visitedNodes: 7424 nodes
  depth: 12 plies
```

Listing 6.11 – OFMC output

6.8 Conclusion

Dans ce chapitre, nous avons décrit une solution basée sur l'utilisation de certificats qui permet d'établir des canaux sécurisés entre un utilisateur et une carte. Le protocole défini a été validé à l'aide de l'outil AVISPA ce qui permet d'assurer qu'il est fiable. En utilisant ce protocole, un utilisateur

peut accéder aux cartes en toutes confiance et échanger des données sans crainte qu'elles puisent être déchiffrées par des entités non autorisées.

Chapitre 7

Quelques outils supplémentaires

Sommaire

7.1 Introduction

Pour une exploitation aisée du cluster Java Card, nous avons mis en place quelques utilitaires permettant d'aider les utilisateurs à déployer et à exécuter leurs applications. Ces outils constituent la quatrième couche de l'architecture logicielle définie dans la section 4.4.3. Dans un premier temps, nous avons développé trois utilitaires qui sont : un outil de composition de services ; un outil d'installation d'applets ; un ordonnanceur d'exécution tolérant aux pannes. Cette liste pourra bien entendu croître au fur et à mesure que de nouveaux besoins apparaîtront.

7.2 Un ordonnanceur tolérant aux fautes

Le calcul parallèle constitue aujourd'hui une des principales utilisations des grilles. En effet, disposer d'un nombre important de ressources permet d'exécuter une quantité importante de processus (ou tâches). Toutefois, il est nécessaire de mettre en place un environnement logiciel permettant d'assurer des mécanismes fondamentaux tels que la réservation de ressources, l'ordonnancement de tâches, etc.

La fonction d'ordonnancement consiste à trouver, pour une tâche donnée, le support d'exécution (le processeur) le plus approprié. Dans le cas d'un ordonnanceur basique, ceci revient à étudier la charge du système (autrement dit de chaque processeur), afin de choisir la ressource la moins chargée pour exécuter la tâche suivante. Le choix de la ressource appropriée peut également dépendre de contraintes matérielles, de la priorité de la tâche, etc., ce qui permet de définir des ordonnanceurs plus complexes.

7.2.1 Un modèle d'ordonnancement pour le cluster Java Card

Nous proposons ici un modèle simple pour l'ordonnancement, mais qui fournit en revanche des mécanismes pour la tolérance aux pannes. Nous nous plaçons dans le contexte où les différentes tâches à exécuter sont indépendantes les unes des autres et qu'elles n'interagissent pas entre elles. Ces différentes tâches sont traitées par une même applet qui doit être installée sur toutes les cartes formant la grille.

Concrètement, l'exécution d'une tâche pour une carte à puce revient à traiter une commande APDU. La fonction d'ordonnancement consiste à observer les différentes ressources, et dès qu'une carte a fini d'exécuter une commande, on lui envoie une nouvelle. La fin de l'exécution d'une tâche (en l'occurrence une commande APDU) est reconnue par la réception d'une réponse APDU de la carte concernée. L'ensemble des commandes à exécuter est exprimé par l'intermédiaire d'une API dédiée. L'ordonnanceur prend alors en charge la répartition et l'envoi des différentes commandes aux cartes disponibles.

Par ailleurs, nous avons mis en place des mécanismes pour la tolérance aux pannes. Quand une carte devient indisponible, la commandes APDU qui lui a été envoyée est retransmise à une autre carte. Pour cela nous avons mis en œuvre un module pour le contrôle des tâches affectées afin de surveiller l'exécution de chaque commande APDU jusqu'à sa terminaison, c'est-à-dire jusqu'à la réception d'une réponse APDU de la carte exécutant la commande. Dans le cas où le traitement d'une commande est interrompue, suite par exemple à l'arrachage de la carte qui l'exécutait, cette commande sera renvoyée à une ressource disponible (en l'occurrence une autre carte).

7.2.2 Les fautes considérées

Les erreurs que nous considérons sont essentiellement d'origine matérielle. Nous présentons dans ce qui suit quelques exemples de fautes traitées :

- défaillance d'une carte : ceci se produit suite par exemple à un problème d'alimentation, de connexion, etc.
- arrachage d'une carte : une carte peut être arrachée par erreur ou même de façon malveillante. Il en est de même pour le débranchement d'un lecteur de cartes.
- erreurs de communication : il s'agit des erreurs de communication entre le client et le ClusterManager mais aussi des communications directes avec les cartes.

7.2.3 Mise en œuvre

Pour mettre en œuvre notre modèle d'ordonnancement, nous avons développé la classe *Job* (listing 7.1) qui correspond à une tâche élémentaire. Cette classe contient les variables `command` et `response`, qui sont des tableaux de `byte` qui représentent la commande et la réponse APDU. Nous définissons également deux variables de type `boolean` : la variable `done` qui permet d'informer si la commande a été traitée et la variable `attributed` qui, lorsqu'elle est à `true`, indique que la commande été déjà envoyée vers une carte et encours d'exécution, c'est-à-dire que carte concernée n'a pas encore envoyée une réponse APDU. L'ensemble des tâches est fourni à l'ordonnanceur sous forme d'une collection d'objets de type `Job`.

L'ordonnanceur doit disposer de l'ensemble des cartes présentes. La liste des ressources disponibles est obtenu via le système de contrôle de l'état de la grille présenté dans la section 5.6. Du point de vue implémentation, une ressource matérielle est représentée par la classe `Resource` qui hérite de `Thread` (listing 7.2). Le traitement principal de cette classe (la méthode `run()`) consiste à récupérer un `Job` pas encore exécuté (depuis la classe *JobScheduler*) et à envoyer la commande APDU correspondante à la carte associée. La classe *JobScheduler* a pour rôle de parcourir la liste des tâches et de retourner un des *Job* qui n'a pas encore été exécuté.

Lorsqu'une carte n'est plus disponible, le `Thread` correspondant est suspendu. La gestion dynamique des ressources est réalisée par la classe *JCardScheduler*, définie dans la classe `SchedulerManager`. La classe *JCardScheduler* implémente l'interface de notification `GridEventListener` (définie dans la section 5.5.1) ce qui permet à l'ordonnanceur d'être informé des événements d'insertion et d'arrachage

```
public class Job {

    byte[] command;
    byte[] response;
    boolean done = false;
    boolean attributed = false;
    [...]
    public byte[] getResponse(){
        return response;
    }
```

Listing 7.1 – Modélisation d'une tâche élémentaire : la classe `Job`

```
public class Resource extends Thread {

    [...]
    JobScheduler jobscheduler;
    Job job;

    /**
    *Récupère un Job et envoie la commande vers la carte tant que le Thread n'est pas suspendu
    **/

    public void run() {

        try{
           [...]
        while (threadSuspended == true)
            wait();
        }
        job = jobscheduler.getJob();
        j.response = proxy.sendAPDU(nameCard, j.cmd);
            [...]
    }
    /**
    *Suspend le Thread
    **/
    public void waitUntil() {

        threadSuspended = true;
    }
    /**
    *éveiller le Thread
    **/
    public void awake() {

        threadSuspended = false;

       }
    [...]
```

Listing 7.2 – Modélisation d'une ressource matérielle : la classe Resource

de cartes. Lorsqu'une carte est insérée, le `Thread` correspondant est réveillé (appel de la méthode `awake()`). Quand'il s'agit d'un arrachage de carte, le `Thread` est suspendu (appel de la méthode `waitUntil()`).

7.3 Composition de services

Afin de mieux gérer les services disponibles dans la grille de cartes à puce, nous avons développé un outil semi-automatique de composition de services qui permet d'orchestrer l'exécution des Card Services d'une application.

```
public class SchedulerManager {

   Resource[] allThread;

   JCardScheduler notif = new JCardScheduler();


   public SchedulerManager(byte[] AID, List<Job> listJobs) {

        [...]
               //S'enregistrer auprès du gestionnaire des événements de la grille
               proxy.susbscribeUser(notif);

        }


    public Ressource searchResource(String aReader, String szCard) {

      [...]

    }

    [...]


    private class JCardScheduler implements NotifyUser {

        Resource aThread;

        public void insertedCard(String szReader, String szCard) {
            aThread = searchResource(szReader, szCard);
            aThread.awake();

        }

        public void removedCard(String szReader, String szCard) {
            aThread = searchResource(szReader, szCard);
            aThread.waitUntil();
        }
[...]
```

Listing 7.3 – Gestion des ressources dynamique par l'ordonnanceur

7.3.1 Intérêt d'une orchestration de services

Comme nous l'avons expliqué dans le chapitre 4, nous considérons la grille comme une agrégation de ressources logicielles représentées par les Card Services. À l'instar des architectures Service Web [20], nous proposons un modèle pour la gestion et l'utilisation des Card Services basé sur les trois étapes fondamentales que sont la publication, la découverte et l'invocation de services. Notre objectif est de fournir une méthode d'accès aux applications carte qui soit beaucoup plus facile que le passage par des APDU.

En général, un utilisateur aura besoin d'intégrer plusieurs services au sein de son application, d'où la nécessité d'utiliser le concept de composition de services. Deux scénarios différents de composition sont envisageables :

- si la composition se fait de façon manuelle, alors il sera à la charge du client de réaliser les invocations de services une par une et de sauvegarder les résultats intermédiaires afin d'obtenir finalement le résultat souhaité.
- si la composition se fait de façon automatisée, le client doit seulement préciser la combinaison des services ainsi que leur ordre d'exécution. Le rôle du système est alors de gérer la coordination et l'exécution de ces services.

La première proposition présente des limites significatives. En effet le client, obligé d'invoquer les méthodes une par une, utilise du temps et des ressources matérielles, et peut même se tromper dans le stockage des résultats intermédiaires surtout quand il s'agit d'une composition complexe. La deuxième solution est beaucoup plus simple. Nous avons donc mis en place un orchestrateur de services qui permet de réaliser la composition de services de manière automatique ou au moins semi-automatique.

7.3.2 Mise en œuvre de la composition de services dans le cluster de Java Cards

L'outil de composition que nous allons présenter a été développé par Monia Ben Brahim et Faten Baccar dans le cadre de leur projet de fin d'étude. Nous donnons ici une brève description de l'outil mis en place. Pour plus de détails le lecteur pourra se référer à [8, 21]. La composition de services est mise en place à travers trois modules fondamentaux, à savoir :

1. un langage de définition permettant de décrire un processus métier ;
2. un moteur d'orchestration qui assure l'exécution d'un processus défini ;
3. une interface graphique conviviale facilitant l'accès aux services disponibles.

Un utilisateur peut modéliser son processus métier en se basant sur le langage de définition. Cette modélisation consiste à préciser l'ordre d'invocation des opérations offertes par le service ainsi que les éventuelles interactions pouvant exister entre les services. Il peut s'agir par exemple d'une exécution séquentielle, parallèle, la définition de variables intermédiaires ou de messages d'entrées/sorties, etc. Le processus métier est décrit selon une structure bien définie qui est traduite en format XML. Le fichier résultant sert par la suite au moteur d'orchestration qui gère l'exécution de l'application et réalise les invocations de services tout en respectant l'ordonnancement défini dans le fichier de description du processus métier. L'interface graphique permet de faciliter la découverte, l'invocation et la composition des services qui sont disponibles dans la grille.

7.3.2.1 Le langage de définition d'un processus métier

Afin de décrire un processus métier, un utilisateur doit disposer d'un langage de définition bien spécifié. Le langage de définition que nous avons conçu fournit une description abstraite de très haut niveau à travers laquelle on peut décrire l'ordre des exécutions et spécifier les interactions entre les différents Card Services via des scripts d'orchestration. Un script d'orchestration [23] est défini comme une activité simple, c'est-à-dire une étape du procédé durant laquelle une action est exécutée. L'ensemble de ces scripts constitue un processus de composition que l'on appelle aussi processus métier.

Le langage de définition que nous proposons s'inspire du langage BPEL [23, 147], un langage conçu pour les Services Web. Le langage proposé prend en considération les spécifiés et les particularités des cartes Java. En effet, les cartes Java n'acceptent que des types de données simples et de petite taille tels que *boolean*, *byte* et *short*. Les types de données manipulables dans un processus métier sont donc limités à ces types. Les types qui sont relativement grands (tels que les *floats*) ne sont pas autorisés par la technologie Java Card, et donc non définis dans notre langage.

Dans la description du processus métier, les variables d'entrées/sorties sont définies de façon explicite. Par exemple :
<input name="status" type= "byte"/>.
<output name="nbTotal" type= "short"/>.

Pour mettre à jour, ou affecter une valeur à une variable, nous avons défini l'activité *<copy>*. La nouvelle valeur peut être déduite d'une expression, ou provenir de la valeur d'une autre variable, comme illustré par le listing 7.4.

Concrètement, un processus métier correspond à une séquence d'actions ou plus exactement à un flux d'activités faisant intervenir un ou plusieurs Card Services. Il est donc nécessaire de déclarer les services qui interviennent dans le processus de composition. La déclaration d'un service, ou plus précisément d'une opération, se fait à travers la balise *<services>* comme le montre le listing 7.5. Dans la description d'un service, il faut préciser le nom de l'opération, la valeur du champ instruction de la commande APDU correspondant à cette opération, l'AID de l'applet implémentant le service, ainsi que les informations de localisation (le nom ou l'adresse IP de la machine, le nom du lecteur et de la carte contenant cette applet). Ces informations peuvent être récupérées depuis le registre de configuration définie dans la section 5.2.1. Aujourd'hui, ceci n'a pas encore été automatisé.

Une fois déclarée, l'opération peut être appelée. L'invocation d'une opération est représentée par l'entité *<invoke>*, comme présenté par le listing 7.6.

```
<!--Une expression à copier-->

<copy>
    <from expression="getVariable(x)*2"/>
    <to variable="NB_MAXIMAL"/>
</copy>


<!--Une variable à copier-->

<copy>
    <from variable="x"/>
    <to variable="NB_MAXIMAL"/>
</copy>
```

Listing 7.4 – Affectation de variables dans le processus métier de composition de Card Services

```
<services>
   <service name="op1" server="147.210.8.203"
       reader="OMNIKEY CardMan 3x21 0@card1"
       Aid="appelt1@4261736963436l6C63756C" ins="4"/>
   <service name="op2" server="147.210.8.203"
       reader="OMNIKEY CardMan 3x21 0@card1"
       Aid="applet2@1161436963433l6C63756C" ins="6"/>
   <service name="op3" server="147.210.8.207"
       reader="OMNIKEY CardMan 3x21 1@card2"
       Aid="applet@4261736963436l6C63756C" ins="4"/>
</services>
```

Listing 7.5 – Déclaration des services dans le processus métier de composition de Card Services

```
<invoke service="op1">
   <inputVariable>status</inputVariable>
   <outputVariable>nbTotal</outputVariable>
</invoke>
```

Listing 7.6 – Invocation d'une opération dans le processus métier de composition de Card Services

En pratique, le processus de composition fait intervenir plusieurs services. Il est donc essentiel de préciser l'ordre d'invocation des Card Services. Dans notre langage nous définissons des flots de contrôle séquentiels et parallèles. L'enchaînement ordonné de l'exécution des actions d'un processus est représenté par l'activité *<sequence>*, qui permet d'englober des activité d'invocation, d'affection ou d'exécution parallèle (représentée par l'activité *<flow>*). Les activités *<flow>* peuvent aussi contenir des activités de type *<sequence>* qui s'exécutent en parallèle (listing 7.7) et qui sont éventuellement liées par des synchronisations.

```
<flow>

   <sequence name ="flow1">
      [.....]
   </sequence>
    <sequence name ="flow2">
      [.....]
     </sequence>
</flow>
```

Listing 7.7 – La structure parallèle dans le processus métier de composition de Card Services

Le mode de synchronisation qui a été adopté est de type producteur-consommateur : un service (consommateur) doit attendre l'exécution d'une autre opération (producteur) afin de pouvoir s'executer. Chaque lien *<link>*, dans une activité *<flow>*, rend une activité cible (activité *<target>*) dépendante d'une activité source (*<source>*), comme le montre le listing 7.8.

```
<flow>
   <links>
      <link nameVar="c" />
   </links>

   <sequence name ="flow1">
     <invoke service="add">
        <inputVariable>s</inputVariable>
        <outputVariable>r1</outputVariable>
        <source linkNameVar="c"/>
     </invoke>
   </sequence>

   <sequence name ="flow2">
      <copy>
        <from expressin=getVariable(c)>
        <to variable ="d" />
        <target linkNameVar="c" producer="flow1" />
      </copy>

      <invoke service="doubler">
        <inputVariable>d</inputVariable>
        <outputVariable>res</outputVariable>
      </invoke>
   </sequence>
</flow>
```

Listing 7.8 – Synchronisation entre blocks d'activité dans le processus métier de composition de Card Services

Notre langage de définition permet également de décrire des structures de contrôle variées (*if*, *for*, *switch*, *repeat*). Ces structures de données sont intégrées dans les flots de contrôle parallèles et séquentiels en utilisant des activités appropriées telles que *<if>*,*<while>*, etc. *<if>*) et des boucles (*<while>*, *<for>*).

7.3.2.2 L'interface graphique

Afin de faciliter l'utilisation des services hébergés dans le cluster Java Cards, nous avons construit une interface graphique déployée côté client permettant de faciliter la découverte et l'invocation des Card Services disponibles. Elle se présente comme une fenêtre divisée en deux parties. La partie à gauche permet d'afficher la liste des services disponibles sous forme d'une arborescence à 3 niveaux. Le premier niveau contient le nom du lecteur et la carte insérée dans ce lecteur. Le deuxième niveau montre le nom de chaque applet et son AID, et le dernier niveau affiche les noms des méthodes de l'applet concernée. L'utilisateur peut sélectionner un service afin d'en afficher une brève description. La partie à droite permet à l'utilisateur d'afficher le description d'un service en XML, d'écrire en XML la description d'un processus métier, ou de charger la description d'un processus afin de l'exécuter. L'interface graphique, présenté à la figure7.1, contient également un menu qui permet de lancer l'exécution du processus de composition chargé dans la partie droite. Une boîte de dialogue s'affiche alors permettant à l'utilisateur de saisir les paramètres d'entré de son processus.

7.3.2.3 Le moteur d'orchestration

Une fois le processus de composition décrit, il faut l'exécuter ; c'est le rôle du moteur d'orchestration. Il analyse le fichier de description du processus de composition et le traduit en un coordinateur de services.

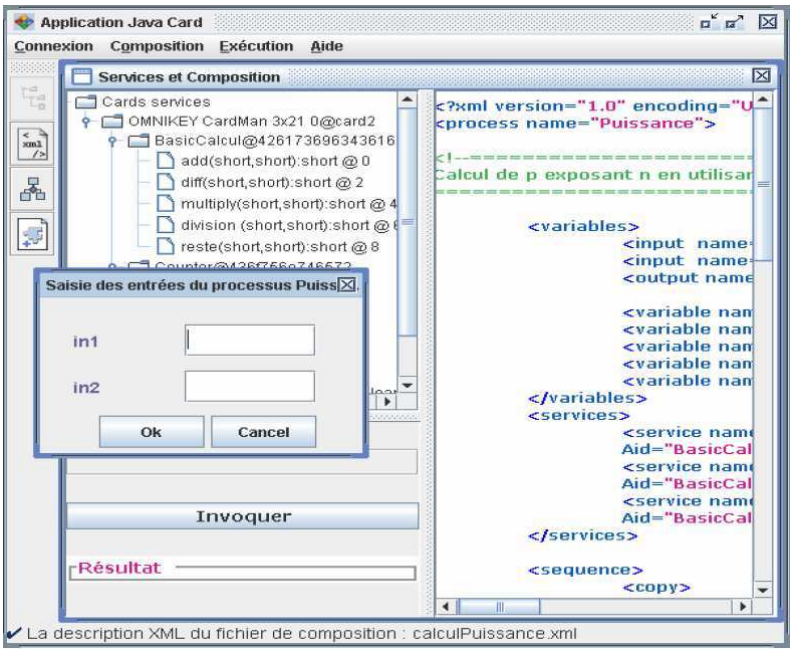


FIG. 7.1 – Interface graphique pour l'exécution d'un processus métier de composition

Le moteur d'orchestration traite la description du processus métier de manière récursive et génère le code de l'application cliente. Le code généré doit être conforme à la description du point de vue de l'ordre d'exécution et des interactions entre les opérations objets de la composition. Le moteur d'orchestration transforme les entités *<invoke>* en appel du service concerné. Il crée dynamiquement des *Threads* Java pour les exécutions parallèles et utilise le modèle producteur/consommateur pour implémenter les synchronisations entre *Threads*.

À la fin de l'exécution de l'application sur les cartes (c'est-à-dire l'exécution des Card Services), le moteur d'orchestration convertit le résultat final de byte en le type spécifié pour le paramètre de sortie du processus.

De plus, nous cherchons à assurer une exécution complète en dépit de la survenue d'erreurs. Pour cela nous avons mis en place des mécanismes de tolérance aux pannes.

7.3.3 Composition de services et tolérance aux pannes

Cette partie a été réalisé dans le cadre des travaux de recherche effectués par Monia Ben Brahim dans le cadre de son Master de recherche [21]. Nous présentons ici les principaux résultats obtenus. Afin d'assurer la continuité de service, une couche logicielle permettant de proposer des mécanismes de tolérance aux fautes est introduite. Cette couche est mise en œuvre à travers un gestionnaire de recouvrement des pannes qui gère la substitution du service défaillant (suite par exemple à l'arrachage de la carte qui l'héberge) par un ou plusieurs autres Services Card. Deux modes de recouvrement sont possibles :

- recouvrement par réplication : ceci consiste à remplacer un service défaillant par un service sémantiquement et syntaxiquement équivalent. Autrement dit, le service défaillant est remplacé par une autre instance, du même service, disponible sur une carte fonctionnelle. La commande APDU est renvoyée vers cette nouvelle carte. S'il reste des commandes initialement destinées au service défaillant, elles seront également redirigées vers le nouveau service.
- recouvrement par composition : ceci consiste à remplacer le service défaillant par une composition de Card Services.

Cependant pour pouvoir réaliser des mécanismes de tolérance aux fautes, il faut disposer d'un outil de recherche automatique permettant de trouver un service ou une composition équivalente à un

service donné. Ce module a donc été développé. Il permet, lors de la défaillance d'un Card Service, de découvrir dynamiquement des services identiques et/ou les compositions éventuelles permettant de le remplacer. L'ensemble des services disponibles est stocké au niveau d'un registre global qui gère les localisations des Card Services dans la grille, ainsi que leurs équivalences sémantiques (en termes de duplication et de compositions). Les services sont répertoriés par groupe au niveau du registre par groupe, suivant la fonctionnalité fournie, afin de faciliter et d'accélérer la recherche des services substituants. Lors d'un éventuel recouvrement, la priorité est donnée aux duplicatas du même service ce qui permet d'obtenir un temps de recouvrement moins important qu'une composition. Il faut noter que la définition d'une composition nécessite une intervention humaine. L'utilisateur doit en effet fournir au registre global les compositions possibles pour un service donné. De plus le registre global doit s'enregistrer auprès du système de contrôle d'état afin d'être informé de la disponibilité des services pour mettre à jour l'ensemble des services qu'il contient.

Un autre module est le *Message Monitor*, dont le rôle est de surveiller les invocations de services élémentaires. Ce composant permet, en cas de problème d'exécution, de récupérer les exceptions et de faire appel au gestionnaire de recouvrement afin de substituer un service défaillant. Il s'agit des fautes présentées dans la section 7.2.2 et qui vont levées une exception lors de l'exécution.

Nous présentons à la figure 7.2 les différentes étapes de l'exécution d'un fichier de composition. Le moteur d'orchestration permet de traduire une composition en invocations de services élémentaires, qui seront transmises vers le cluster de cartes. Le module *Message Monitor* permet de contrôler l'exécution des services et lorsqu'une exception est levée, il fait appel au gestionnaire de recouvrement. Ce dernier, en interrogeant le registre global, réalise les traitements adéquats : le service défaillant est remplacé soit par une service identique soit par une composition équivalente. La nouvelle exécution sera aussi réalisée sous le contrôle du *Message Monitor*. Lorsqu'il s'agit d'une exécution sans erreur, la réponse APDU est directement envoyée vers le moteur d'orchestration.

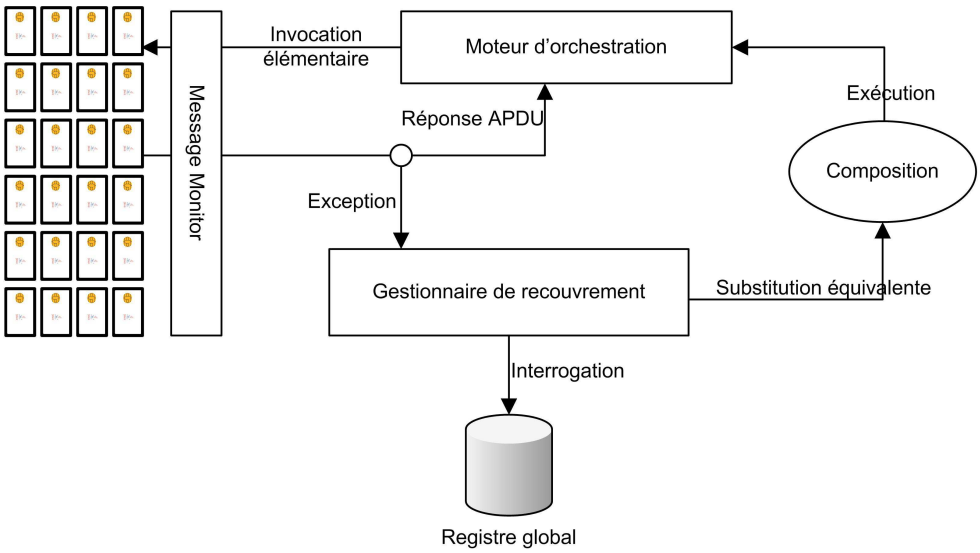


Fig. 7.2 – Exécution avec recouvrement d'une composition

7.4 Outil d'installation

Une fois la plate-forme cluster Java Card mise en place, un des premiers problèmes rencontrés fut l'installation des applications sur les différentes cartes constituant la grille. Notre objectif était de trouver un moyen qui permettrait d'installer un ensemble d'applications sur plusieurs cartes à la fois.

Plusieurs outils de développement pour les cartes Java existent. La plupart de ces outils offrent des fonctionnalités de base conforme à GlobalPlatform, notamment l'authentification mutuelle avec la carte et l'installation/désintallation des applications.

Les outils commerciaux : les outils commerciaux sont principalement proposés par les fabricants de cartes. Chaque fabricant développe son propre outil qui est commercialisé avec ses cartes et qui est destiné à une utilisation exclusive. Ces outils offrent plusieurs fonctionnalités pour la gestion des cartes à travers un environnement de développement accessible depuis une interface graphique conviviale, et souvent un simulateur intégré. Parmi ces outils, on trouve sur le marché l'outil *GemXpresso* Rad pour les cartes *Gemplus*, *Sm@rtCafé* de *Giesecke & Devrient*, *Cyberflex* de *Schlumberger*, etc. Ces différents outils ne sont pas compatibles entre eux, c'est-à-dire il n'est pas possible d'utiliser un outil proposé par un fabricant avec des cartes d'un autre fabricant. Comme tout produit commercial, le code de ces outils n'est pas libre, et n'est pas sous une licence publique.

Les outils gratuits : le kit de développement de *Sun* est l'un des premiers outils gratuits qui ait été conçu pour les cartes Java. Il permet de créer des applets et de les compiler. L'inconvénient de cet outil est qu'il fonctionne en ligne de commande, ce qui n'est pas très convivial. Mais son défaut principal est qu'il n'implémente pas les spécifications GlobalPlatform.

Depuis peu de temps, des outils gratuits tels que *GPShell* [140], pour la plupart *open source*, sont apparus permettant d'offrir à l'utilisateur l'ensemble quasi complet des commandes GlobalPlatform. Parmi les fonctionnalités offertes on trouve l'installion/désintallation des applications, la récupération de la liste des paquetages chargés, etc. Ces outils permettent en quelque sorte de surmonter l'inconvénient des outils commerciaux, à savoir l'incompatibilité des produits. En effet, ils peuvent être utilisés avec n'importe quelle carte, qu'elle provienne d'un fabricant ou d'un autre. Néanmoins, ces kits requirent un minimum de configuration afin de pouvoir gérer plusieurs modèles. Lorsqu'on souhaite changer le type de carte qu'on manipule, il faut modifier les paramètres de configuration relatifs aux clés statiques, l'AID du gestionnaire de carte, etc.

Cependant et malgré leur diversité, aucun de ces outils ne répondait à nos besoins pour une utilisation dans le contexte du cluster de cartes. Les outils évoqués précédemment ont été conçu pour un fonctionnement en local alors que nous nous intéressons à une utilisation distante. Notre objectif était d'offrir à l'utilisateur la possibilité de lancer le processus d'installation sur la grappe de cartes depuis sa propre machine. De plus, les outils existants ne permettent de gérer qu'une seule carte à la fois, alors que nous cherchions à réaliser une installation en parallèle, c'est-à-dire à installer une ou plusieurs application sur des cartes différentes de manière simultanée. Nous avons donc développé un outil propre au cluster Java Card et qui répondait à nos besoins.

L'outil de développement pour la grille Java Card comporte deux parties. La première partie est une API respectant les spécifications GlobalPlatform et qui offre les commandes fondamentales définies dans le standard. La deuxième partie est un outil graphique convivial qui permet, en utilisant les API GlobalPlatform de la première partie, d'accéder à distance au cluster Java Card, et de paralléliser le processus d'installation/désintallation d'un ensemble d'applications sur plusieurs cartes.

7.4.1 Installation conforme à *GlobalPlatform*

Comme nous l'avons expliqué dans le chapitre 2, les applications chargées sur la carte portent l'extension *.cap*. Il s'agit d'une représentation binaire de type archive permettant de décrire les différents composants constituant un package java (les imports, les classes, les méthodes, etc).

Ce format de fichier permet de stocker un ensemble de composants de façon indépendante. Chaque composant est dans un fichier individuel dont le nom a pour extension *.cap*. Le format de chacun de ces fichier suit le format TLV (*Tag, Length, Value*) présenté au tableau 7.1 :

Étiquette	Taille	Données

TAB. 7.1 – Format du fichier d'un composant du paquetage

Afin d'embarquer le fichier sur la carte, il faut disposer d'un analyseur pour en extraire les différentes parties et construire les commandes APDU appropriées permettant le chargement de l'application sur la carte. Cette génération de commandes APDU doit être conforme aux spécifications GlobalPlatform. Celles-ci définissent les valeurs des champs d'instructions, la structure des champs de

données, etc. Le fichier *.cap* est donc traduit en suite de commandes APDU qui sont envoyées vers le gestionnaire de carte, qui réalise le chargement effectif du package sur la carte.

7.4.2 Les API *GlobalPlatform*

Cette partie a été réalisée par un groupe d'étudiants dans le cadre d'un projet de fin d'année à l'ENSEIRB (École Nationale Supérieure d'électronique, Informatique et Radiocommunications de Bordeaux). L'API qui a été développée permet de gérer le processus d'installation en respectant les recommandations GlobalPlatform. En analysant un fichier *.cap*, les commandes adéquates pour charger une application sont générées. GlobalPlatform définit trois types de commandes relatives à l'installation et au chargement des applications qui sont *InstallForInstall*, *Load* et *InstallForLoad*. Un analyseur de fichiers permet d'extraire les différents parties constituant le fichier *.cap* et de construire les commandes appropriées. D'autres commandes de base décrites dans les spécification GlobalPlatform, sont été aussi intégrées dans cette API, notamment celles qui concernent l'authentification mutuelle (*Initialize Update* et *External Authenticate*). Le tableau 7.2 présente l'ensemble des opérations implémentées dans l'API développée.

7.4.3 Interface graphique

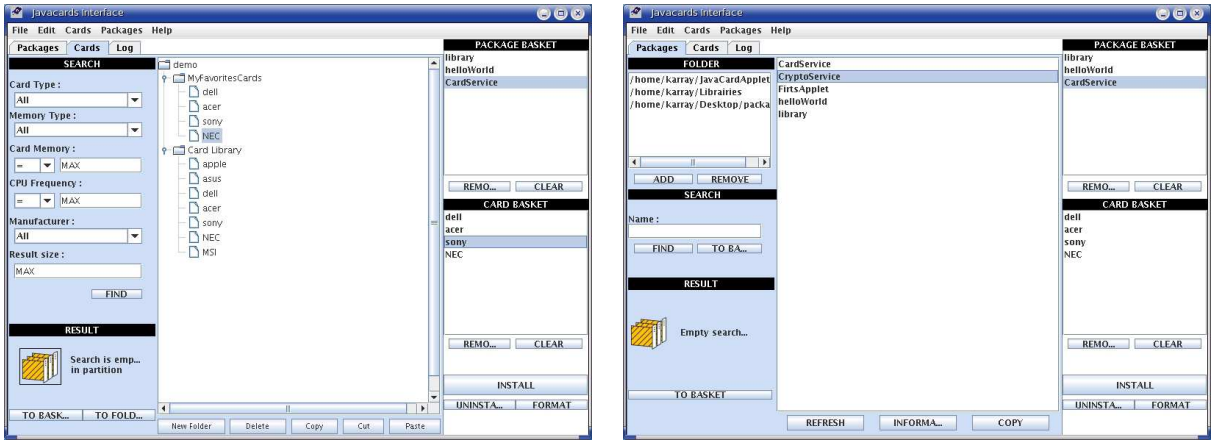
Cette partie a été réalisée par un groupe d'étudiants dans le cadre d'un projet de fin d'année à l'ENSEIRB (École Nationale Supérieure d'électronique, Informatique et Radiocommunications de Bordeaux). À l'état actuel, l'outil n'est pas tout à fait fonctionnel et nécessite quelques modifications de code pour qu'il soit opérationnel.

7.4.3.1 Présentation générale

Nous avons mis en place un environnement graphique permettant d'installer les applications sur différentes cartes de la grille. Il permet de :

- donner à l'utilisateur la possibilité de choisir l'application à installer (un fichier *.cap*),
- présenter à l'utilisateur la liste des cartes disponibles sur la grille afin qu'il puisse choisir les cartes sur lesquelles il souhaite installer son application ;
- procéder à l'installation de l'application choisie en parallèle sur les différentes cartes sélectionnées en utilisant l'API déjà développée.

La figure 7.3 présente l'interface graphique de l'outil d'installation qui comprend :



(a) l'onglet *Cards* (b) l'onglet *Packages*

Fig. 7.3 – L'interface graphique pour l'outil d'installation

commande	*Initialize Update*
Fonctionnement	Il s'agit de la première étape dans le processus d'authentification mutuelle
commande	*External Authenticate*
Fonctionnement	C'est la deuxième commande de l'authentification mutuelle
commande	*InstallForLoad*
Fonctionnement	Cette commande permet de prévenir le gestionnaire de carte pour qu'il prépare la carte à charger un paquetage
commande	*Load*
Fonctionnement	Permet de charger les données contenues dans un paquetage, représenté par un fichier *.cap*
commande	*InstallForInstall*
Fonctionnement	Crée une instance de l'application contenue dans le paquetage chargé
commande	*Select*
Fonctionnement	Sélectionne une applet, à laquelle les commandes suivantes seront destinées
commande	*getStatus*
Fonctionnement	permet de retourner les applications de la carte sous la forme d'une liste AID
commande	*Delete*
Fonctionnement	Supprime un paquetage ou une application. Un paquetage ne peut être supprimé que si toutes les applets qui l'utilisent ont été déjà effacées

TAB. 7.2 – Liste des principales commandes GlobalPlatform

- une partie droite permettant de présenter les paniers de travails, *Package Basket* et *Card Basket* (voir section 7.4.3.2), et qui permettent de lancer l'installation/désinstallation des paquetages contenus dans le panier des applications, sur l'ensemble des cartes figurant dans l'autre panier (panier cartes) ;
- une partie gauche présentant trois onglets permettant de gérer les cartes, les paquetages et de consulter un journal d'événements :
  - lorsque l'onglet *Cards* est choisi, la liste des cartes apparaît au centre. Les cartes disponibles sont affichées une forme arborescence dans de différents groupes. Un outil de recherche est disponible à gauche dans l'interface permettant de faire des requêtes de recherche sur les cartes disponibles (l'interface (a) de la figure 7.3).
  - quand l'onglet *packages* est activé, la liste des applications (fichiers *.cap*) est affichée au centre de l'interface. Le menu de gauche permet de garder une liste des dossiers récemment explorés (l'interface (b) de la figure 7.3).
  - l'onglet journal (*log*) permet tout simplement de visualiser l'historique des actions effectuées par l'utilisateur.

7.4.3.2 La solution des paniers

Afin d'offrir une gestion par lot, la solution des paniers a été retenue. Nous avons défini deux types de paniers : un panier de cartes (*Card Basket*) et un panier de paquetages (*Package Basket*). L'utilisateur est donc amené à sélectionner les applications qu'il désire installer en les glissant dans le

panier associé. Il peut alors sélectionner les cartes sur lesquelles il souhaite les installer (l'ordre de ces étapes n'a pas d'importance). Une fois les paniers remplis, il est possible d'installer ou de désinstaller les paquetages sélectionnés sur les cartes retenues.

7.5 Conclusion

Nous avons présenté dans ce chapitre les outils et le utilitaires que nous considérons comme essentiels pour le déploiement et l'exécution des applications. Ils facilitent l'exploitation des cartes mises à disposition. Ces différents outils constituent la quatrième couche de l'architecture logicielle définie dans la section 4.4.3. D'autres outils peuvent aussi être développés selon les besoins des utilisateurs.

Chapitre 8

Quelques applications de validation

Sommaire

8.1 Introduction

Pour valider la plate-forme grille de cartes à puce, nous avons développé un certain nombre d'applications nécessitant du calcul distribué et/ou sécurisé. La première application (simpliste) que nous avons mise en œuvre consiste à calculer la fractale de Mandelbrot en utilisant les Java Cards. La seconde application s'inscrit dans un contexte réel, celui de la fouille de données. À travers la grille de cartes à puce, nous apportons des solutions aux problèmes de sécurité pour des applications de ce type. Finalement, la troisième application met en évidence la capacité de calcul parallèle fournie par la grille de cartes à puce, en particulier pour des applications de type calcul itératif asynchrone.

8.2 La fractale de *Mandelbrot*

Elle a été développée avec le concours des membres de l'équipe Systèmes et Objets Distribués dont je cite les noms Pascal Grange, Damien Sauveron, Pierre Vignéras et moi même.

8.2.1 Définition mathématique

Découverte par Benoît Mandelbrot dans les années 80, la fractale qui porte son nom est l'une des plus étudiées [19]. Mathématiquement parlant cette fractale est définie comme suit. Considérons le

plan complexe $\mathcal{P}$, où ;

$$\forall (x,y) \in \mathbb{R}^2, P(x,y) \in \mathcal{P} \equiv z = x + yi, z \in \mathbb{C}$$

Pour tout point $z \in \mathbb{C}$, on définit la suite $M(z)$:

$$M(z) = \begin{cases} z_0 = z, \\ z_{n+1} = z_n^2 + z \quad n \in \mathbb{N} \end{cases} \tag{8.1}$$

La fractale de Mandelbrot est définie comme étant l'ensemble des points $z \in \mathbb{C}$ pour lesquels la suite $|M(z)|$ est bornée.

8.2.2 Algorithme

Pour représenter la fractale de Mandelbrot, on utilise une variation de couleur en fonction du nombre d'itérations permettant de tester la convergence de la suite $M(z)$ lorsque l'on vérifie si le point considéré appartient ou non à l'ensemble de Mandelbrot. Si la suite ne diverge pas au bout d'un grand nombre d'itération fixé, on considère qu'il fait partie de la fractale de Mandelbrot.
Pour chaque point du plan complexe $\mathcal{P}$, l'algorithme effectue des itérations successives selon l'équation 8.1 afin de décider si le point se situe à l'intérieur ou à l'extérieur de l'ensemble de Mandelbrot. Pour afficher le résultat des calculs, on parcourt l'image initiale en spécifiant les valeurs en abscisse et ordonnée, et pour chaque pixel, l'algorithme retourne une couleur qui dépend du nombre d'itérations qui correspond au test de convergence de la suite $M(z)$ en ce pixel.

Le principe de l'algorithme consiste à calculer pour chaque point $z \in \mathbb{C}$ la valeur *count(z) n* (équation 8.2). Si le point considéré est dans l'ensemble de Mandelbrot, alors il n'admet aucune valeur n pour laquelle $|z_n| \geq 2$. Pour cela, il faut tout d'abord choisir un nombre d'itérations maximal en fonction duquel la suite $M(z)$ appliquée en ce point est considérée convergente. Ainsi, nous définissons un nombre d'itérations maximal (*bailout*) B pour décider si le point fait partie de l'ensemble de Mandelbrot. Autrement dit, si $\forall i < B$, on a $|z_i| < 2$ alors nous considérons que le point appartient à l'ensemble de Mandelbrot. La fonction $count(z)$ est définie comme suit :

$$count(z) = \begin{cases} n, & \text{if } \forall i < n,\ |z_i| < 2 \text{ and } |z_n| \geq 2 \\ B, & \text{if } \forall i \leq B,\ |z_i| < 2 \end{cases} \tag{8.2}$$

Pour représenter l'ensemble de Mandelbrot, an assigne à chaque pixel du plan complexe $P(x,y)$ d'affixe $z = x + iy$ une couleur $color(z)$ qui dépend de $count(z)$. En particulier, si $count(z) = B$, la couleur $color(z)$ est le noir. De plus, il faut choisir une surface minimale S qui correspond à la surface de la région qui sera traitée par une unique carte. Nous disposons pour cela d'une méthode appelée `discrepancy()` permettant de subdiviser la figure initiale jusqu'à l'obtention de régions de résolution plus petite que S. Ceci fait, il faut boucler sur tous les points de cette surface, en utilisant la méthode `calculateAll()` pour leur appliquer l'algorithme de calcul qui déterminera leurs appartenance ou non à l'ensemble de *Mandelbrot*. La figure 8.1 présente le fonctionnement général de l'algorithme.

8.2.3 Mesure de performance

Pour exécuter l'application *Mandelbrot* sur un cluster de cartes, nous avons développé des applets Java Card qui implémentent les méthodes `discrepancy()` et `calculateAll()`. Un premier problème technique est apparu à ce niveau : le type de données *double* n'étant pas supporté par la plate-forme Java Card, il a fallu développer une bibliothèque adéquate. Toutefois, vues les restrictions matérielles des cartes, et en particulier le peu de mémoire disponible, l'implémentation d'une telle bibliothèque s'est avérée extrêmement délicate. De plus, nous avons dû installer un proxy au niveau du Cluster-Manager permettant de traduire les appels de méthodes `discrepancy()` et `calculateAll()` en des commandes APDU à envoyer aux cartes comme le montre la figure 8.2.

Nous avons exécuté notre application pendant plusieurs heures sur des Java Cards en faisant varier le nombre de cartes à chaque test. Il faut noter ici que l'application Mandelbrot a été développée à

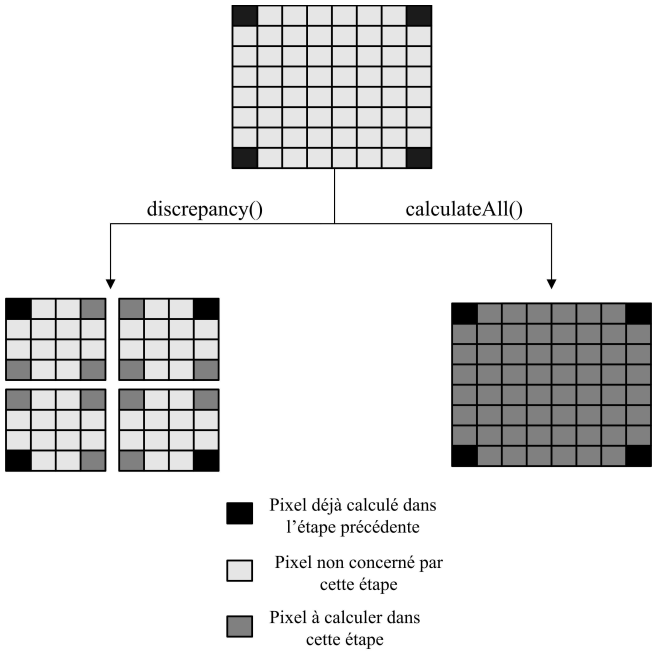


Fig. 8.1 – Principe de l'algorithme de calcul de l'ensemble de Mandelbrot

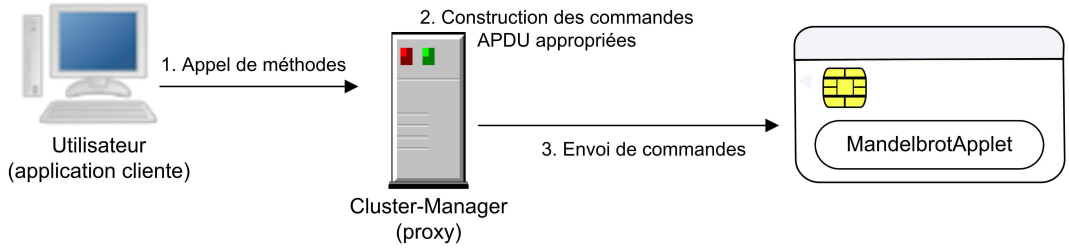


Fig. 8.2 – Schéma d'exécution de l'application Mandelbrot sur les cartes

l'époque où nous ne disposions que d'une plate-forme matérielle qui ne supportait que 9 cartes ; ce nombre correspond donc au nombre maximal de cartes utilisées dans les benchmarks.

Par ailleurs, après plusieurs itérations, nous avons constaté que les cartes commençaient à se détériorer étant donné le nombre considérable d'écritures sur la mémoire non volatile que nécessite l'application. Pour cette raison, nous avons décidé d'arrêter les tests à une certaine étape. Aussi, pour cette même raison, nous avons jugé qu'il ne sera pas judicieux de reprendre les tests avec la nouvelle plate-forme matérielle lorsqu'elle était disponible ce qui nous permettrait d'utiliser un nombre plus important de cartes pour les mesures de performance.

Les benchmarks ont été réalisés sur des cartes provenant de différents fabricants. Dans un premier temps, pour chaque test effectué, nous avons utilisé que des cartes du même modèle. Nous avons ensuite testé l'application sur un mélange de cartes constitué de 3 cartes *GemXpresso Pro R3 E32PK*, 2 *JCOP31bio*, 2 *GemXpresso Pro R3 E64PK* et 2 *SmartC@fé Expert*. La figure 8.3 représente une capture d'écran de l'exécution de l'application *Mandelbrot*. À cause d'un *bug* de programmation dans l'implémentation initiale de la bibliothèque *double* sur les cartes, il y a quelques points erronés sur la figure. Mais, cela n'est pas très gênant pour les tests réalisés et ne met pas en cause les benchmarks puisque le bug apparaît sur toutes les cartes de façon régulière.

Les résultats des mesures sont présentés dans le tableau 8.1. La fractale initiale possède une surface de 200x200 pixels. Les paramètres de l'algorithme sont B = 20 (nombre d'itérations maximal) et S = 100 pixels (surface minimale de la région à partir de laquelle on pourra calculer la suite $M(z)$). Le

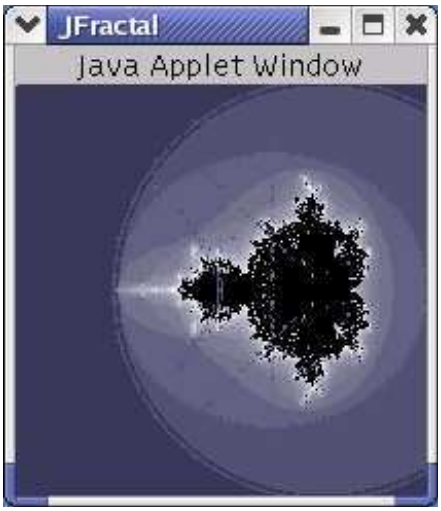


FIG. 8.3 – Résultat de l'exécution de l'application *Mandelbrot* sur les cartes

nombre de cartes varie entre 5 et 9. À force de répéter les tests, certaines cartes ont atteint le nombre maximal de cycles d'écriture mémoire, quelques tests n'ont pas donc été réalisés, ce qui explique le manque de quelques valeurs dans le tableau 8.1.

Modéle	Nombre de cartes	Temps d'exécution (en *mn*)
GemXpresso Pro R3 E32PK	9	197
	7	232
	5	325
JCOP31bio	9	252
	7	321
	5	432
GemXpresso Pro R3 E64PK	9	377
Sm@rtCafé Expert	9	390
Mélange	9	345

TAB. 8.1 – Les différents bancs d'essai de l'application *Mandelbrot* en mode non sécurisé

Nous avons également effectué quelques mesures en utilisant un canal sécurisé *SCP01* tel que défini dans GlobalPlatform. La sécurité ne concerne donc que les commandes APDU envoyées du lecteur vers la carte. Nous avons toutefois constaté que le surcoût engendré est pratiquement négligeable. Les cartes que nous avons utilisées disposent d'un co-processeur cryptographique dédié et le temps de calcul des fonctions cryptographiques est infime : il est d'environ 100 μs pour un *DES*. Or, selon les spécifications GlobalPlatform implémentées sur les cartes, le canal sécurisé était basé sur des fonctions *Triple DES*, ce qui explique que le temps de traitement des commandes relatives à la sécurité soit très petit.

Nous avons repris les mêmes *benchmarks* sur des cartes de type *Fujitsu* (cartes qui nous ont été prêtées comme produit de test pour réaliser les *Benchmarks*) et les résultats obtenus sont meilleurs que pour la première série de tests (tableau 8.2).

Par exemple, l'exécution de l'application sur un ensemble de 9 cartes *Fujitsu* n'a duré que 135 mn par rapport à 197 mn sur des cartes *Gemplus GemXpresso Pro R3 E32PK* (le meilleur temps dans la première série de tests). Ceci est dû à la différence de technologie entre les cartes. Dans la première série de tests, les cartes que nous avons utilisées possédaient une mémoire persistante de type EEPROM alors que les cartes *Fujitsu* disposent d'une mémoire FRAM. La FRAM présente l'avantage d'être plus rapide lecture/écriture que l'EEPROM. Les FRAM sont caractérisées par un nombre de cycles d'écriture supportés plus grand, qui est de l'ordre de 10^{12}, contre seulement 10^5 pour les EEPROM.

L'application Mandelbrot telle que nous l'avons écrite est plus adaptée pour les cartes possédant une mémoire de type FRAM qu'EEPROM.

Nombre de cartes	Temps d'exécution (en *mn*)
9	135
7	159
5	215
3	358
1	1021

Tab. 8.2 – Les différents bancs d'essai de l'application *Mandelbrot* pour les cartes *Fujitsu*

Rappelons ici que le point que nous souhaitons traiter, avec le cluster Java Card, est celui de la sécurité. Ainsi, la faible vitesse d'exécution de l'application Mandelbrot sur les cartes Java doit être relativisée, cette application ne servant qu'à valider l'aspect fonctionnel de la plate-forme.

8.3 Une application de type fouille de données (*Data-Mining*)

8.3.1 Contexte

Les applications de type fouille de données consistent à analyser une quantité importante de données afin d'en déduire des informations utiles. Généralement, l'entité qui effectue la fouille et le propriétaire des données sont distincts. Par conséquent, des problèmes de sécurité se présentent. Il peut être essentiel de garantir les deux propriétés suivantes :

- la confidentialité des données traitées ;
- l'intégrité de l'exécution des algorithmes et la confidentialité des critères de fouille.

Nous nous basons sur les propriétés sécuritaires de la grille de cartes à puce pour garantir ces contraintes.

L'exemple que nous traitons est le suivant. Un acteur gouvernemental (par exemple le le bureau des douanes et de la protection des frontières des États-Unis ; le CBP : *Customs and Borders Protection*) souhaite effectuer une recherche dans les informations concernant les passagers des vols à destination de son territoire. Pour cela, il souhaite accéder aux fichiers des passagers des compagnies aériennes opérant les vols concernés (par exemple Air France).

8.3.2 Principe

La situation est donc la suivante. Pour des raisons de sécurité, le CBP souhaite analyser les fichiers des passagers d'Air France pour certains vols. Air France (AF dans la suite) pour sa part ne souhaite pas communiquer ses fichiers au CBP pour des raisons de confidentialité des données de ses passagers, mais surtout à cause des lois en vigueur qui lui interdisent de donner accès à ces fichiers [51]. Le CBP ne souhaite pas non plus divulguer les critères mis en œuvre dans son analyse des passagers. Les caractéristiques de l'application AF-CBP sont les suivantes :

- elle est de type *data-mining*, c'est-à-dire que l'on dispose d'un jeu de données que l'on souhaite analyser pour en extraire des informations ;
- les données ou une partie d'entre elles doivent rester confidentielles (les fichiers passages de AF) ;
- le code manipulant les données doit rester confidentiel (le code du CBP).

Le prototype que nous avons implémenté prend ces contraintes en considération.

Pour pouvoir distribuer les informations concernant les passagers sur les différentes cartes, nous avons mis en place une application permettant la saisie des passagers. À chaque passager est associée une clé unique permettant à la compagnie aérienne de l'identifier. C'est d'ailleurs cette clé qui est utilisée par le CBP pour identifier un éventuel suspect sans obtenir l'identité réelle du passager,

information considérée confidentielle. Une API, développée par Air France est installée sur l'ensemble des cartes afin de fournir un accès restreint aux aux données considérées comme non confidentielles.

De son côté, le CBP déploie son application de fouille sur les différentes cartes. Pour effectuer l'analyse des données, le code de CBP accède aux fichiers des passagers à travers l'API développée par Air France. Le code du CBP, quand il identifie un passager suspect, renvoie la clé concernée vers une carte particulière qui permet de centraliser les résultats de la fouille. Le CBP peut alors entamer un phase de négociation avec Air France afin de récupérer toutes les informations (y compris les données confidentielles) concernant les passagers identifiés comme suspects.

8.3.3 Mise en œuvre

La première étape de la mise en œuvre consiste à charger les informations des passagers sur l'ensemble des cartes de la plate-forme. Pour cela, nous avons développé pour la compagnie Air France un outil graphique (présenté à la figure 8.4) lui permettant de saisir la liste de ses passagers.

FIG. 8.4 – Interface de saisie des passagers

Un passager est représenté par la classe `Passenger`. Cette classe contient toutes les informations le concernant : son nom, son prénom, sa nationalité, etc. Nous avons aussi développé et installé sur chaque carte une applet appelée `PassengerManager` permettant à Air France de gérer la liste de ses passagers comme illustré sur la figure 8.5.

L'applet `PassengerManager` fournit également une interface réduite permettant au CBP de récupérer les informations concernant un passager. Cette interface n'autorise l'accès qu'aux données qu'Air France a spécifiées non confidentielles. Pour cela, la classe `PassengerManager` implémente l'interface `PassengerView` qui est une interface *Shareable*. Lorsqu'une classe est définie comme partagée, elle autorise des applets (ou toute autre classe) de contextes différents à invoquer les méthodes définies dans son interface partageable sans qu'elles ne soient bloquées par le pare-feu. Un tel mécanisme est réalisé sous le contrôle du JCRE. À travers les méthodes définies dans l'interface `PassengerView`, la classe `PassengerManager` fournit un accès restreint aux fichiers des données des passagers. Autrement dit, seules les données considérées come non confidentielles pourront être consultées par des applications extérieures, et en particulier par l'application CBP (l'applet `PassengerChecker`) comme présenté dans

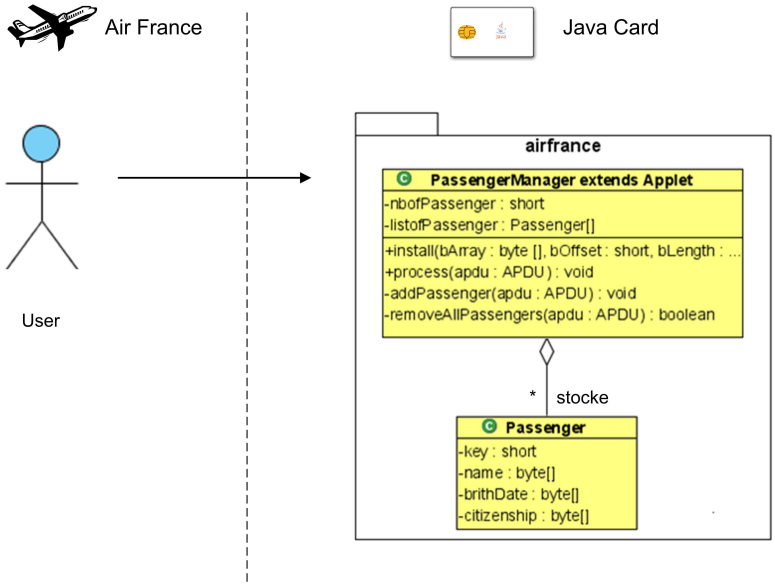


Fig. 8.5 – Architecture de gestion des passagers sur la carte

la figure 8.6. La compagnie Air France partage ainsi avec le CBP les données relatives à ses passagers selon une procédure sécurisée et contrôlée par le JCRE.

Pour déterminer quels sont les passagers qui apparaissent comme suspects, le CBP spécifie des critères de recherche ou de fouille. Un outil graphique présenté à la figure 8.7 lui permet de saisir les paramètres qui seront utilisés pour rechercher d'éventuels suspects. Une fois les critères spécifiés, ils sont transmis aux instances de l'applet `PassengerChecker`, représentant le CBP sur les différentes cartes. Le Cluster-Manager envoie alors aux différents cartes une commande APDU contenant les paramètres de recherche afin d'appeler la méthode d'analyse sur toutes les cartes disponibles comme le montre la figure 8.8.

Chaque applet `PassengerChecker` analyse alors les informations des passagers enregistrées sur sa carte afin de trouver ceux qui correspondent aux critères de recherche. Chaque fois qu'un suspect est trouvé, son identifiant *key* est récupéré et envoyé vers une applet particulière installée sur la carte nommée `CBPServer`. Cette carte a pour rôle de centraliser tous les résultats de fouille comme le montre la figure 8.9. Cette étape illustre le fonctionnement du mécanisme de la proactivité forte présentée dans la section 5.5.2.1.

Enfin, quand toutes les cartes ont fini leur recherche locale, le CBP envoie à l'applet `SuspectManager` (présente sur la carte que nous appelons `CBPServer`) une commande prédéfinie pour obtenir la liste des éventuels suspects (figure 8.10). Il entame alors une procédure de négociation avec la compagnie Air France pour que cette dernière lui communique toutes les informations utiles (y compris éventuellement les données confidentielles) sur les passagers suspects correspondants aux différentes clés (*key*) trouvées.

8.4 Cluster Java Card pour le calcul parallèle

8.4.1 Contexte

Le calcul parallèle distribué consiste à subdiviser un problème initial en sous-problèmes appelés tâches élémentaires ou processus. Ces différentes tâches sont exécutées sur différentes machines distantes ce qui permet d'avoir un traitement plus rapide qu'une exécution séquentielle. Les architectures de type grille favorise ce type de calcul :disposant d'un nombre important de ressources matérielles, on peut exécuter un grand nombre de processus de façon simultanée ce qui permet de fournir un capacité de calcul parallèle importante. Ceci peut être appliqué par exemple pour la calcul de la trajectoire

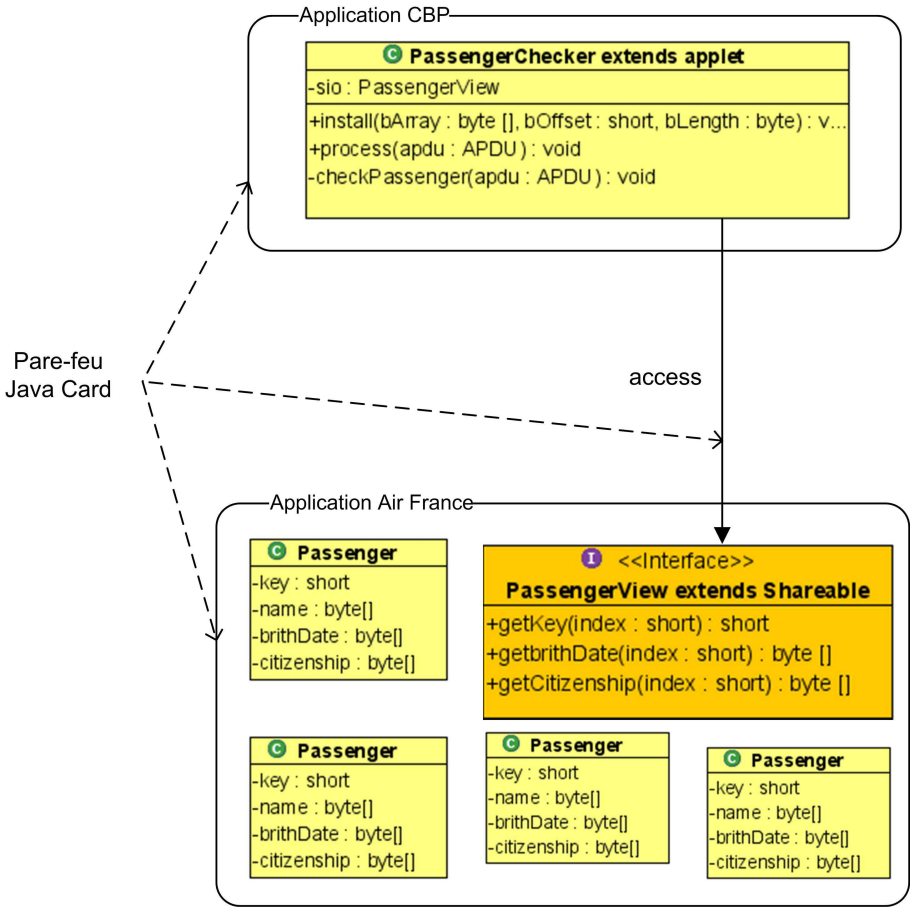


FIG. 8.6 – Partage des fichiers des passagers dans la Java Card

FIG. 8.7 – Interface de saisie de critéres de recherche

optimale, la recherche de clé cryptographique, etc. Nous nous intéressons ici à un cas particulier de traitement particulier qui consiste à exécuter un même programme sur différents ressources avec des données en entrée différentes.

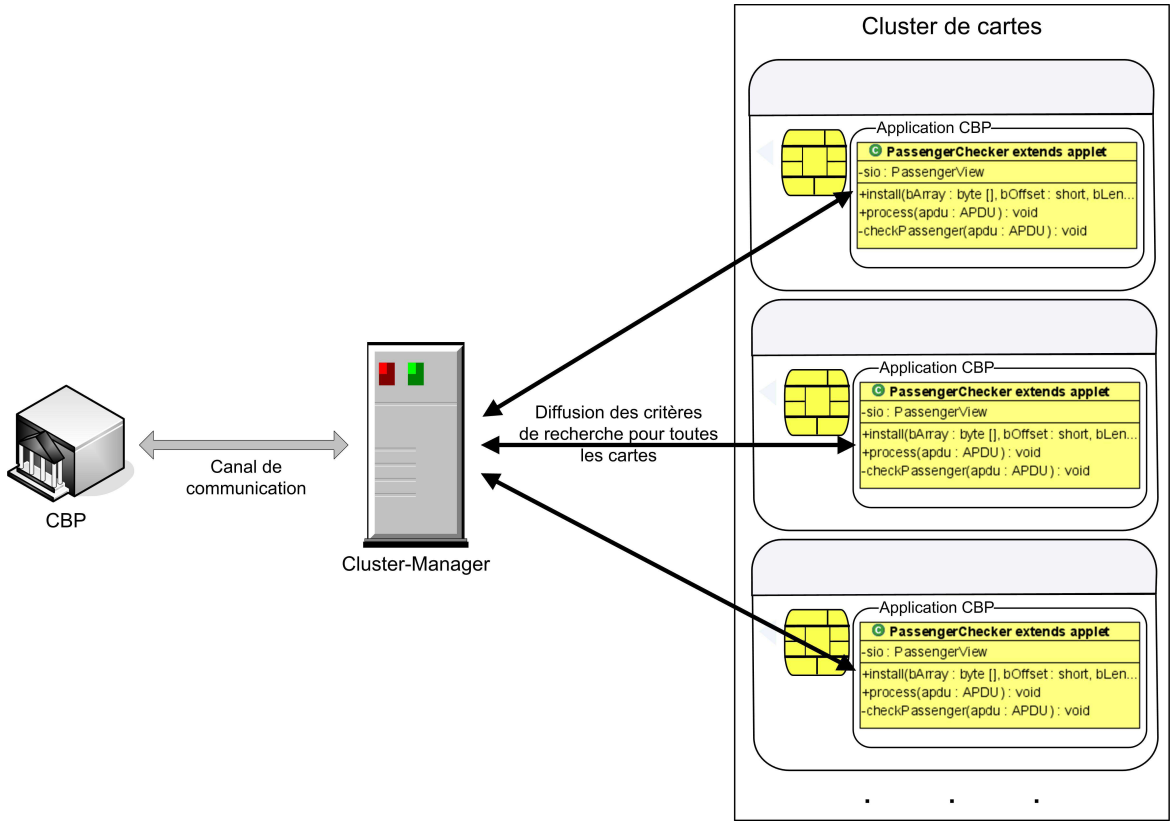


Fig. 8.8 – Schéma de diffusion de la requête d'analyse aux applets *PassengerChecker*

8.4.2 Un exemple d'application de conversion d'images

8.4.2.1 Principe

L'application présentée ici consiste à convertir une image couleur en une image en niveaux de gris en utilisant un cluster de cartes Java. Pour cela, l'image de départ est subdivisée en sous-images (fragments) de dimension déterminée ($n * m$ pixels). Les différents fragments formant l'image initiale sont envoyés vers l'ensemble des cartes disponibles pour effectuer les transformations nécessaires. Pour atteindre cet objectif, nous avons dû développer plus de 250 lignes de code (y compris le code nécessaire pour l'ordonnancement) réparties en 11 classes. De plus, nous nous sommes basés sur les fonctionnalités fournies par la couche intergiciel, notamment en ce qui concerne l'aspect volatil (connexion et déconnexion) des cartes.

La couleur d'un pixel est exprimée dans l'espace de couleur que nous utilisons est le RVB (Rouge, Vert, Bleu). Ces valeurs sont envoyées à la carte qui calcule le niveau de gris correspondant selon la formule :

$$gray = \frac{\max(red, green, blue) + \min(red, green, blue)}{2}$$

Dès qu'un fragment est traité, il est récupéré aussitôt par l'application cliente, qui se trouve sur un PC, afin de construire au fur et à mesure l'image finale.

À travers cette application, nous ne cherchons pas à effectuer les calculs de conversion de manière efficace, ils peuvent en effet être facilement réalisés sur un simple PC. Notre objectif est d'appliquer l'ordonnancement tolérant aux fautes décrit dans la section 7.2.

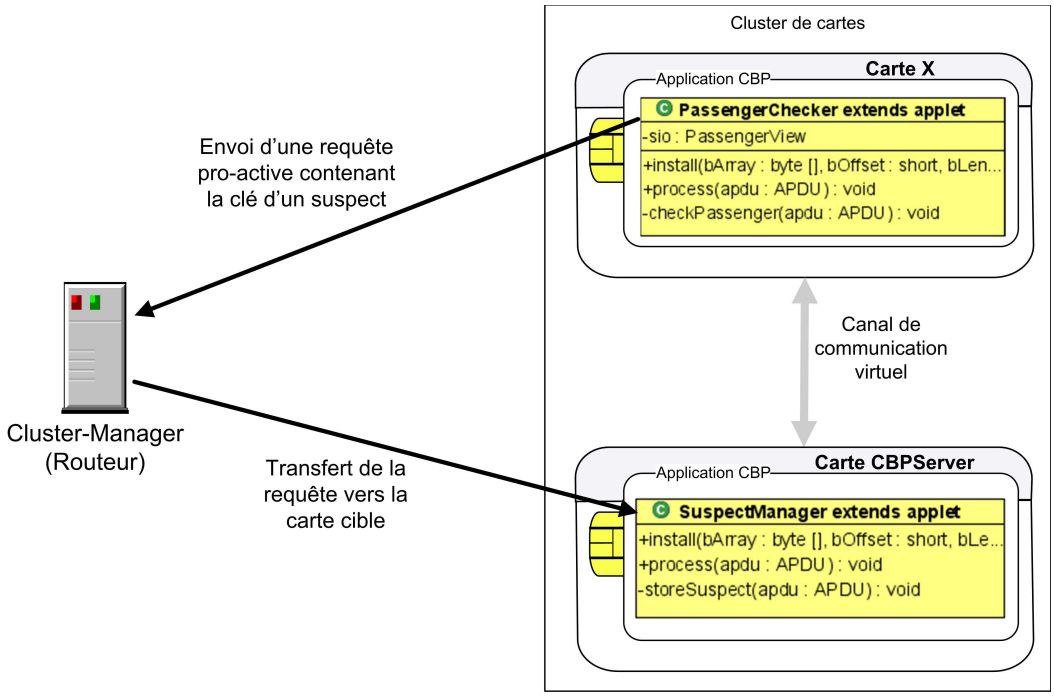


FIG. 8.9 – Schéma de retour des résultats

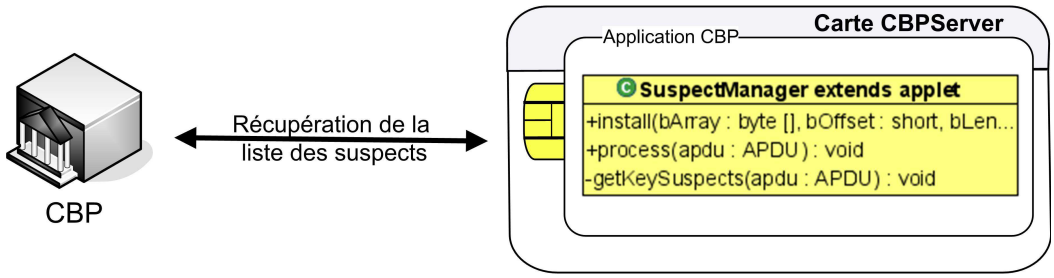


FIG. 8.10 – Schéma de récupération de la liste de clés des suspects trouvés

8.4.2.2 Mise en œuvre

La première étape consiste à découper l'image initiale en sous-images (fragments) faisant chacune 64 pixels. Les pixels de chaque fragment sont encapsulés dans des commandes APDU. Une fois toutes les commandes construites, elles sont fournies à l'ordonnanceur dans des objets de type *Job*. L'ordonnanceur dispatche alors les différentes commandes APDU vers les cartes disponibles. Sur chacune des cartes, nous avons déployé une applet qui convertit un pixel couleur en niveaux de gris selon la formule de la section 8.4.2.1.

Chaque fois qu'un fragment est traité, le résultat d'exécution est récupéré par l'application cliente et l'image finale est affichée au fur et à mesure. Deux modes d'affichage ont été mis en place. Le premier mode consiste à représenter l'avancement global de l'image finale, c'est-à-dire l'évolution de la construction de l'image en niveaux de gris comme le montre la figure 8.11. Dans le deuxième mode, nous nous intéressons au fonctionnement de chaque carte. Nous avons pour cela développé une interface graphique, présentée à la figure 8.12, qui permet de suivre les traitements effectués par chaque Java Card.

Cette interface graphique présente les lecteurs. Son organisation reflète l'emplacement physique des lecteurs dans le cluster. Si le lecteur contient une carte, les fragments traités par cette dernière sont affichés. Lorsque la carte est arrachée, l'image est réinitialisée. Bien entendu, quand il n'existe

aucune carte insérée dans le lecteur, aucune image n'est affichée sur le lecteur associé. L'image finale représentée sur la figure 8.11 est obtenue par superposition des différentes images affichées sur la figure 8.12.

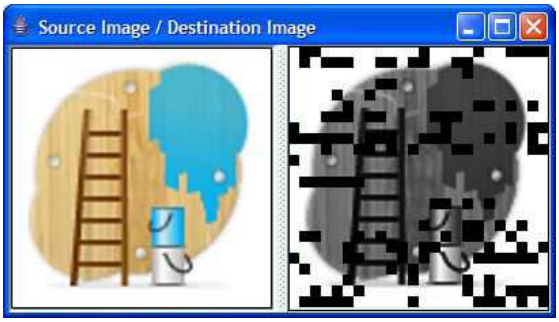


Fig. 8.11 – Évolution générale de la conversion de l'image

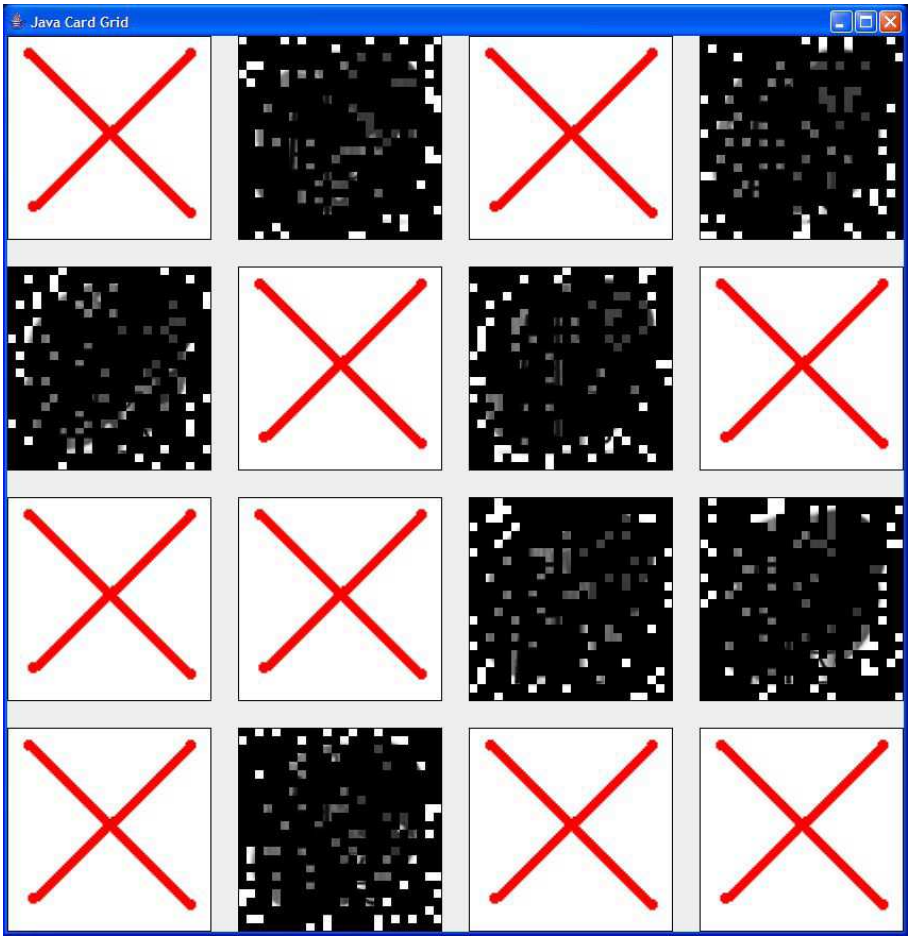


Fig. 8.12 – Évolution de l'activité des cartes pour la conversion de l'image

Dans le contexte de grille de cartes à puce, nous nous intéressons principalement aux applications dont l'aspect sécurisé prime sur l'aspect calcul.

8.4.3 Application du modèle itératif asynchrone

Dans certains problèmes, un programme initial peut être décomposé en tâches élémentaires qui sont exécutées sur des processeurs différents. Il est possible que les tâches soient parfois dépendantes : certains algorithmes locaux peuvent avoir besoin des résultats d'exécution d'autres tâches. Les méthodes dites itératives font partie de ce type de calcul. Dans les algorithmes itératifs une itération se déroule en deux phases. La première consiste en l'exécution en locale de la tâche élémentaire. La seconde phase est une phase de communication dans laquelle le résultat des calculs locaux est diffusé aux autres processeurs. Ces résultats ainsi diffusés sont agglomérés par les tâches les recevant à leurs résultats locaux. On continue à itérer jusqu'à terminer l'application ou à atteindre un certain seuil de convergence. Parmi les classes des algorithmes itératifs, nous distinguons les algorithmes itératifs asynchrones [104]. La figure 8.13 illustre le schéma général des algorithmes asynchrones. Les itérations sont représentées par des rectangles. L'envoi de résultats est schématisé par des petits flèches après chaque itération.

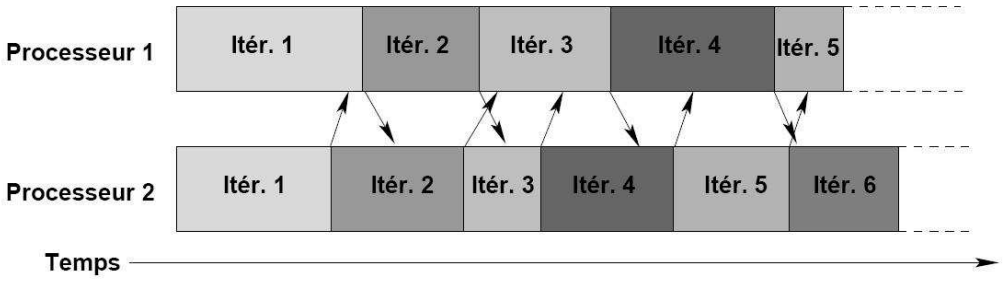


FIG. 8.13 – Exemple d'algorithme itératif asynchrone (source [104])

De façon abstraite, les algorithmes itératifs asynchrones suivent le modèle présenté au listing 8.1.

```
Initialisation des données locales

repeat
   Calcul des données locales
   Envoi/Réception des résultats
   Mise à jour des données locales
   Test de convergence local
until convergence globale
```

Listing 8.1 – Principe des algorithmes itératifs asynchrones

Dans notre exemple, chaque carte calcule un fragment donné de l'image initial. Ensuite elle diffuse aux autres cartes le numéro du fragment concerné. La diffusion des résultats est assurée par le Cluster-Manager. Quand celui-ci reçoit une réponse d'une carte quelconque, il examine le champs SW de la réponse afin de vérifier s'il s'agit d'une réponse de diffusion, ou d'une simple réponse comme le montre le listing 8.2.

Dans le cas de diffusion, il renvoie la commande contenue dans la réponse de la carte vers les autres cartes disponibles. La commande diffusée contient le numéro du bloc que la carte vient de traiter, ce qui évite que d'autres cartes recalculent le même fragment. Les cartes mettent à jour la liste des blocs calculés.

Ce processus est réitérée jusqu' à ce qu'une carte détecte que tous les blocs ont été traités. Elle envoie alors aux cartes, toujours à travers Cluster-Manager, une commande dans laquelle elle indique le fin de l'application.

```
public class SmartCard {

    //correspond au type Card définie dans JPC/SC
    Card theCard;
    [...]

    public byte[] sendAPDU(byte[] C_APDU) {

        byte[] R_APDU;

        R_APDU = theCard.Transmit(C_APDU, 0, C_APDU.length);

        //Analyse du champ SW de la réponse APDU
        if(R_APDU[R_APDU.length-2]==(byte)0x9F){

            //extraire la commande à diffuser depuis la réponse APDU
            byte [] Broadcast_C_APDU = new byte[R_APDU.length-2];
            System.arraycopy(R_APDU,0,Broadcast_C_APDU, 0, Broadcast_C_APDU.length);

            //Diffusion de la commande APDU
            SCardGridManager.broadcastAPDU(Broadcast_C_APDU);
        }
        [...]
    }
    [...]
```

Listing 8.2 – Gestion de diffusion des résultats locaux

8.5 Conclusion

La sécurité constitue aujourd'hui un aspect fondamental de beaucoup d'applications. Elle a pour objectif d'assurer la confidentialité des codes et des données ainsi que l'intégrité de l'exécution des applications. Nous avons présenté dans ce chapitre quelques prototypes d'applications. Le but étant de faire la preuve de concept et de la plate-forme grille de cartes à puce. Avec l'évolution des capacités des cartes, il sera possible de se baser sur des architecture de ce type afin de mettre en place des applications réelles pour des domaines où l'aspect sécurité est essentiel, notamment dans le *data-mining*.

Chapitre 9

Conclusion

Ce document est le fruit des travaux de recherche menés au LaBRI (Laboratoire Bordelais de Recherche en Informatique) à l'Université Bordeaux I. Nos travaux de recherche ont porté sur la conception, la mise en œuvre et la validation d'un environnement logiciel pour le calcul distribué sécurisé sur une grille de cartes à puce. En terme de sécurité, nous cherchons à garantir la confidentialité du code et l'intégrité de l'exécution de l'application, mais aussi la sécurité des échanges entre l'utilisateur et la carte. Une de nos contributions est la proposition d'un protocole de communication basé sur l'utilisation de certificats numériques permettant de protéger les messages échangés entre la carte et l'utilisateur.

Nous avons aussi proposé une solution au problème de la passivité de la carte. Nous avons simulé un schéma de communication pro-actif dans lequel la carte peut appeler un code en dehors de son environnement d'exécution (contrairement à une utilisation classique).

Une autre contribution concerne l'introduction de la notion de *Card Service* dans le but de faciliter l'intégration des cartes à puce dans les systèmes distribués qui est aujourd'hui remarquablement absente dans de telles architectures. L'utilisation des *Card Services* permet de communiquer avec une application embarquée sur la carte de façon transparente tout en offrant une abstraction vis à vis des détails de communication de bas niveau. Un outil de composition approprié a été également conçu permettant à l'utilisateur de faire une composition de plusieurs services en une seule requête.

La plupart des points abordés ont fait l'objet de publications [22, 30, 32, 33, 34, 53] dans des conférences scientifiques. Nous avons également réalisé d'autres travaux de recherche, toujours en rapport avec la technologie des cartes à puce, liés au thème de l'externalization des ressources de la carte et qui ont fait aussi l'objet de publication [31]. Dans ces travaux, nous proposons un mécanisme de *swap* de données sécurisé ce qui permet de disposer d'une capacité mémoire beaucoup plus importante que la mémoire native.

En ce qui concerne les perspectives des travaux de recherche réalisés durant cette thèse, nous envisageons d'adopter le concept de grille de cartes à puce pour les cartes SIM des téléphones portables. Une autre perspective est d'utiliser la puissance de traitement parallèle fournie par la grille et les capacités des calculs cryptographiques de la carte afin de concevoir et de développer un algorithme distribué pour le décèlement de clés.

Annexe A

Cycle de vie de la carte

Nous présentons dans ce qui suit les étapes de la vie d'une carte et les différentes acteurs qui interviennent.

Les acteurs

Au cours de sa vie, une carte à puce passe principalement par les acteurs suivants :

- Le fabricant de puce ou le fondeur : c'est lui qui fabrique le circuit intégré (plus précisément des galettes de silicon contenant la puce) à partir du silicium. Les principaux fabricants de puces sont Fujitsu, Philips, ST Microelectronics, etc.
- Le fabricant de carte : il produit une carte vierge par assemblage du circuit intégré et du support plastique. Parmi les fabricants de cartes, nous pouvons citer Gemplus, Axalto, Oberthur, Giesecke & Devrient, etc.
- L'émetteur de cartes : c'est lui qui propose la carte aux utilisateurs ; il s'agit par exemple d'une banque. L'émetteur reste le propriétaire des cartes du point de vue légale.
- Le fournisseur d'application : en négociant avec l'émetteur, celui-ci autorise le fournisseur d'applications à utiliser de l'espace mémoire dans ses cartes pout y héberger une ou plusieurs applications. Généralement, l'émetteur et le fournisseur d'applications désignent une même entité.
- L'utilisateur final : la carte sera confiée à son utilisateur final (son porteur) et qui s'en sert dans sa vie quotidienne en appelant les applications qui y sont hébergées.

Le cycle de vie de la carte

1. **La fabrication.** La première étape dans la fabrication de la carte est la phase de développement du masque, appelée opération de masquage. Elle consiste à construire les composants matériels de la puce : le microprocesseur, les différentes mémoires, etc. Elle est effectuée par le fabricant des puces (le fondeur) qui choisit la technologie appropriée pour son produit.

   Par la suite le fabricant de cartes découpe les galettes de silicium (*silicon wafer*) qu'il reçoit du fondeur afin de fabriquer le micro-module. Dans le cas des cartes avec contacts, le micro-contact sera assemblé sur la puce durant cette étape.

   L'étape suivante est appelée opération d'encartage et consiste à insérer (assembler) le micro-module sur le support plastique. Les cartes à puce sont ensuite testées afin d'éliminer celles qui sont mal formées après l'encartage. Le produit obtenu est une carte vierge qui pourra être utilisée.
2. **Le cycle de vie.** La première étape dans le cycle de vie de la carte est la phase d'initialisation qui consiste à inscrire dans sa mémoire persistante des données communes aux applications relatives à l'émetteur. Cette étape est réalisée par le fabricant de cartes selon la demande et les besoins de l'émetteur. L'étape suivante est la personnalisation qui permet de mettre sur la carte des

informations personnelles concernant son utilisateur final. Il est possible par exemple d'inscrire des données personnelles (un code secret, une clé d'authentification) et de personnaliser le corps plastique (en imprimant le nom de son utilisateur, sa photo, etc.). La carte peut alors délivrée à son porteur. Généralement après une certaine durée d'utilisation la carte sera invalide. Il est aussi possible qu'elle soit volée, ou perdue, ou que sa mémoire soit saturée ce qui constitue sa fin de cycle de la vie.

Annexe B

Une application d'administration à distance du cluster Java Card

L'application d'administration à distance constitue un outil de supervision permettant à l'utilisateur (en l'occurrence l'administrateur) de contrôler l'état fonctionnel des lecteurs et des cartes formant le cluster Java Card. Cette application permet à un administrateur distant de visualiser la topologie de la grille et de déterminer l'état des lecteurs, c'est-à-dire de voir quels sont les lecteurs libres et ceux qui contiennent une carte. Afin de repérer un lecteur donné, il est nécessaire de trouver une correspondante entre le nom attribué au lecteur par PC/SC et son emplacement dans le cluster. On souhaite que le nom du lecteur indique sa position (C1, C3, etc.), comme le montre la figure B.1, afin de pouvoir l'identifier.

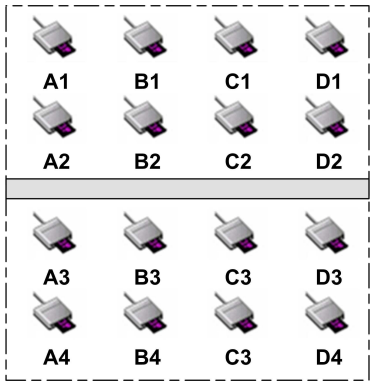


Fig. B.1 – Organisation des lecteurs dans le cluster

Le problème de nommage PC/SC

Malheureusement, la façon dont PC/SC attribue les noms de lecteurs est indéterministe. Si on dispose de deux lecteurs identiques, PC/SC leur attribuera des noms de façon complètement aléatoire. De plus, le nom d'un même lecteur peut être différent entre deux sessions PC/SC.

Concrètement, pour attribuer un nom à un lecteur, PC/SC utilise le modèle de ce lecteur auquel il rajoute deux octets attribués de façon aléatoire. Dans le cluster Java Cards, nous utilisons des lecteurs identiques, de même modèle. Il n'est pas possible de déterminer la position d'un lecteur en se basant sur le nom attribué par PC/SC (figure B.2).

Nous nous sommes donc intéressés à résoudre ce problème. L'idée a été d'introduire le numéro de série du lecteur (numéro unique par lecteur) dans le nom attribué par PC/SC. De cette façon, il est

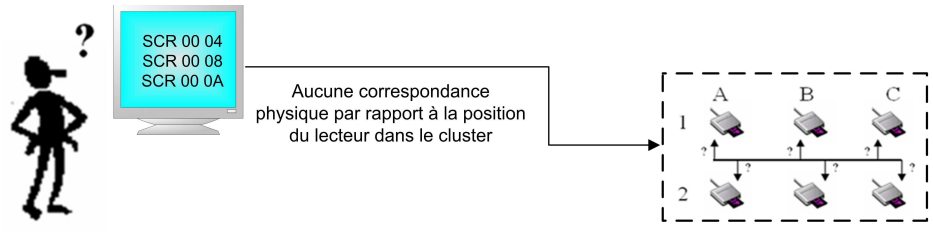


FIG. B.2 – Manière d'attribution des noms par PC/SC

possible d'établir une correspondance entre le lecteur et la position de ce dernier dans le cluster, c'est-à-dire entre le nom attribué par PC/SC et le lecteur physique. Nous avons donc apporté quelques modifications dans le code de PC/SC afin que les noms des lecteurs contiennes le numéro de série du lecteur concerné. Nous avons à cet effet créé un patch (listing B.1) que nous avons envoyé au mainteneur de PC/SC Lite. Cette fonctionnalité a été intégrée dans la version officielle de PC/SC Lite.

```
/*
 * MUSCLE SmartCard Development ( http://www.linuxnet.com )
 *
 * $Id: hotplug_libusb.c $
 *
 */

static struct _readerTracker {
    char status;
    char bus_device[BUS_DEVICE_STRSIZE]; /* device name */
    char *fullName; /* full reader name (including serial number) */
} readerTracker[PCSCLITE_MAX_READERS_CONTEXTS];


LONG HPAddHotPluggable(struct usb_device *dev, const char bus_device[], struct _driverTracker *driver) {

[...]

#ifdef ADD_SERIAL_NUMBER
    if (dev->descriptor.iSerialNumber)
    {
        usb_dev_handle *device;
        char serialNumber[MAX_READERNAME];
        char fullname[MAX_READERNAME];

        device = usb_open(dev);
        usb_get_string_simple(device, dev->descriptor.iSerialNumber, serialNumber, MAX_READERNAME);
        usb_close(device);

        snprintf(fullname, sizeof(fullname), "%s (%s)", driver->readerName, serialNumber);

        readerTracker[i].fullName = strdup(fullname);
    }
    else
#endif
        readerTracker[i].fullName = strdup(driver->readerName);

[...]
```

Listing B.1 – Modification de l'attribution des noms de lecteurs dans PC/SC Lite

L'outil d'administration résultant

Grâce à cette modification de PC/SC, il est maintenant possible de suivre l'évolution de l'état du cluster : toute insertion et tout arrachage d'une carte est signalé à l'administrateur. Les lecteurs sont représentés sur l'interface graphique par des boutons portant comme texte la position du lecteur correspondant dans le cluster. Lorsque le bouton est non fonctionnel, ceci signifie qu'il n'y pas de

carte insérée dans le lecteur. Une fois une carte est insérée, le bouton correspond au lecteur concerné devient fonctionnel. Sur un simple clic, l'administrateur peut alors acquérir des informations sur la carte présente ainsi que sur le lecteur (protocole de communication, nom de lecteur, nom de carte, etc.) comme le montre la figure B.3.

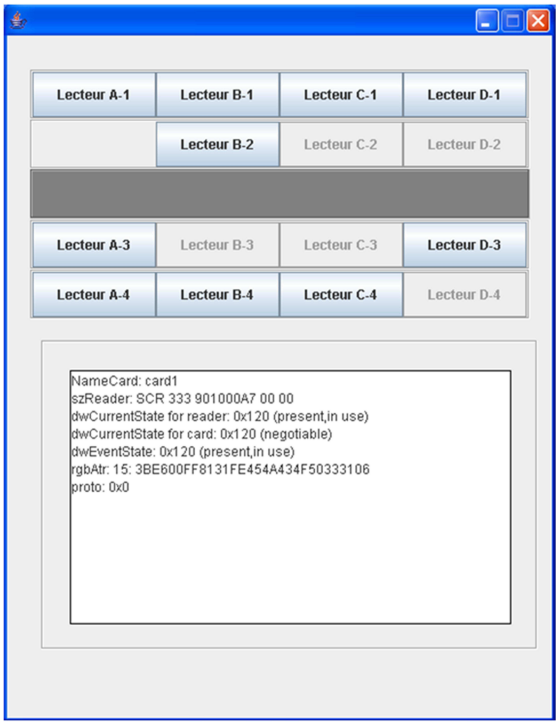


FIG. B.3 – Visualisation de l'état du cluster Java Card chez l'administrateur

Annexe C

Validation Avispa

```
role cvp_Init (User, Card, HTTP_Server : agent,
               Pk_U, Pk_C: public_key,
               Static_Key : symmetric_key,
SndUserCard, RcvUserCard : channel(dy))

played_by User
def=

  local State : nat,
       Card_Challenge, Host_Challenge : text,
        Session_Key : {text.text}_symmetric_key -> symmetric_key,
        Host_Cryptogram : {text.text}_symmetric_key -> text,
        Card_Cryptogram : {text.text}_symmetric_key -> text


  const session_key2, host_cryptogram, card_cryptogram : protocol_id,
        true : bool
  init State := 0

  transition

  1. State = 0 /\ RcvUserCard(start) =|>
     State' := 1 /\ SndUserCard(User.HTTP_Server)


   2. State = 1 /\ RcvUserCard(true) =|>
      State' := 2 /\ Host_Challenge':= new()
                  /\ SndUserCard({Host_Challenge'}_(Pk_C))




   3. State = 2 /\ RcvUserCard({Card_Challenge'}_(Pk_U).
   {Card_Challenge'.Host_Challenge}_({Host_Challenge.Card_Challenge'}_(Static_Key))) =|>
      State' := 3 /\ Session_Key':={Host_Challenge.Card_Challenge'}_(Static_Key)
                  /\ Card_Cryptogram':= {Card_Challenge'.Host_Challenge}_(Session_Key')
                  /\ Host_Cryptogram':= {Host_Challenge.Card_Challenge'}_(Session_Key')
             /\ secret(Session_Key',session_key2 ,{User, Card})
            /\ request (User, Card, card_cryptogram, Card_Cryptogram')
```

```
                 /\ SndUserCard(Host_Cryptogram')
                 /\ witness(User, Card, host_cryptogram, Host_Cryptogram')


   4. State =3 /\ RcvUserCard(true)=|>
       State':=4

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


role cvp_Verify (Authority, HTTP_Server, Card, User : agent,
              Pk_A, Pk_CA : public_key,
               H, Hash : hash_func,
               RcvAuthorityCard, SndAuthorityServer,
               RcvAuthorityServer, SndAuthorityCard : channel(dy))


played_by Authority
def=
 local State   : nat,
       Pk_U  : public_key,
        ReqNum, Signature : text

       const sig, signature  : protocol_id

 init State := 0




  transition

  1. State = 0 /\ RcvAuthorityCard(ReqNum'.User.HTTP_Server)=|>
       State' := 1 /\ SndAuthorityServer(User)

2. State = 1 /\ RcvAuthorityServer(Pk_U.User.{Hash(Pk_U.User)}_inv(Pk_CA))=|>
       State' := 2 /\ Signature':={H(ReqNum.Pk_U)}_inv(Pk_A)
                      /\ SndAuthorityCard(ReqNum.Pk_U.Signature')
                   /\ witness(Authority,Card, signature, Signature')




end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


role getCertificate(HTTP_Server, Authority, User, Card : agent,
               Pk_CA, Pk_U : public_key,
                Hash : hash_func,
                SndServerAuthority, RcvServerAuthority: channel(dy))

played_by HTTP_Server
```

```
def= local
       State : nat


init State := 0

  transition

  1. State = 0 /\ RcvServerAuthority(User)=|>
        State' := 1 /\ SndServerAuthority(Pk_U.User.{Hash(Pk_U.User)}_inv(Pk_CA))


end role



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


role cvp_Resp (Card, User, Authority, HTTP_Server : agent,
       Pk_C, Pk_A : public_key,
       H : hash_func,
              Static_Key : symmetric_key,
              RcvCardUser, SndCardAuthority, RcvCardAuthority, SndCardUser : channel(dy))



played_by Card def=

  local State : nat,
       Host_Challenge,Card_Challenge, ReqNum : text,
       Pk_U : public_key,
       Session_Key : {text.text}_symmetric_key -> symmetric_key,
       Card_Cryptogram : {text.text}_symmetric_key -> text,
       Host_Cryptogram : {text.text}_symmetric_key -> text,
      Signature : text



  const session_key1, signature, host_cryptogram, card_cryptogram : protocol_id,
     true: bool

  init State := 0

  transition

  1. State = 0 /\ RcvCardUser(User.HTTP_Server)=|>
     State' := 1 /\ ReqNum':=new()
                  /\ SndCardAuthority(ReqNum'.User.HTTP_Server)


  2. State = 1 /\ RcvCardAuthority(ReqNum.Pk_U.{H(ReqNum.Pk_U)}_inv(Pk_A))=|>
      State' := 2 /\ Signature' := {H(ReqNum.Pk_U)}_inv(Pk_A)
              /\ request(Card,Authority,signature,Signature')
                   /\ SndCardUser(true)
```

```
   3. State = 2 /\ RcvCardUser({Host_Challenge'}_(Pk_C))=|>
      State' := 3 /\Card_Challenge' := new()
                  /\ Session_Key':={Host_Challenge'.Card_Challenge'}_(Static_Key)
                  /\ Card_Cryptogram':= {Card_Challenge'.Host_Challenge'}_(Session_Key')
               /\ secret(Session_Key',session_key1, {User, Card})
                  /\ SndCardUser({Card_Challenge'}_(Pk_U).Card_Cryptogram')
                  /\ witness(Card,User,card_cryptogram,Card_Cryptogram')



  4. State = 3 /\ RcvCardUser({Host_Challenge.Card_Challenge}_Session_Key) =|>
      State' := 4 /\ Host_Cryptogram':= {Host_Challenge.Card_Challenge}_(Session_Key)
                     /\ request(Card, User,host_cryptogram,Host_Cryptogram')
                     /\ SndCardUser(true)



end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


role session(User, Card, Authority, HTTP_Server : agent,
             Pk_U, Pk_C, Pk_A, Pk_CA : public_key,
             H, Hash : hash_func,
             Static_Key: symmetric_key)
def=

  local SUC, RUC, RCU, SCA, RCA, SCU, RAC, SAS, RAS, SAC, SSA, RSA : channel (dy)

  composition
          cvp_Init(User, Card, HTTP_Server,
                   Pk_U, Pk_C, Static_Key, SUC, RUC)
       /\ cvp_Resp(Card, User, Authority, HTTP_Server,
                   Pk_C, Pk_A, H, Static_Key, RCU, SCA, RCA, SCU)
       /\ cvp_Verify(Authority, HTTP_Server, Card, User,
                   Pk_A, Pk_CA, H, Hash, RAC, SAS, RAS, SAC)
       /\ getCertificate(HTTP_Server, Authority, User, Card,
                      Pk_CA, Pk_U, Hash, SSA, RSA)

end role


role environment()
def=

  const u, c, a, http_server : agent,
        static_key : symmetric_key,
         h, hash2 : hash_func,
      pku, pkc, pka, pki, pkca : public_key

  intruder_knowledge = {u, c, a, static_key, h, hash2, http_server,
                        pki, pku, pkc, pka, pkca, inv(pki)}

  composition

        session(u, c, a, http_server, pku, pkc, pka, pkca, h, hash2, static_key) /\
```

```
       session(u, i, i, http_server, pku, pki, pka, pkca, h, hash2, static_key) /\
       session(i, c, a, http_server, pki, pkc, pka, pkca, h, hash2, static_key)/\

session(u, c, i, http_server, pku, pkc, pki, pkca, h, hash2, static_key)
end role


goal

  %confidentialité des clés de session
 secrecy_of session_key1, session_key2

%Authentification mutuelle
 authentication_on card_cryptogram % de la carte
 authentication_on host_cryptogram % de l'utilisateur

%Authentification d el'entité déléguée
    authentication_on signature

end goal



environment()
```

Nous présentons dans le listing C.1 le résultat de sortie produit par l'outil SATMC.

```
SUMMARY
  SAFE

DETAILS
  STRONGLY_TYPED_MODEL
  BOUNDED_NUMBER_OF_SESSIONS
  BOUNDED_MESSAGE_DEPTH

PROTOCOL
  workfileA10131.if

GOAL
  %% see the HLPSL specification..

BACKEND
  SATMC

COMMENTS

STATISTICS
  attackFound false boolean
  upperBoundReached true boolean
  graphLeveledOff 2 steps
  satSolver zchaff solver
  maxStepsNumber -1 steps
  stepsNumber 2 steps
  atomsNumber 0 atoms
  clausesNumber 0 clauses
  encodingTime 0.03 seconds
  solvingTime 0 seconds
  if2sateCompilationTime 0.37 seconds

ATTACK TRACE
  %% no attacks have been found..
```

Listing C.1 – SATMC output

Glossaire

AID	**Application IDentifier**. Identifiant numérique unique assigné à une application carte.
APDU	**Application Protocol Data Unit**. Unité de données échangée entre la carte à puce et le terminal au niveau de la couche application. Un APDU peut se présenter sous forme de commande, ou de réponse.
Applet	Application écrite pour carte à puce Java.
ASN.1	**Abstract Syntax Notation One**. Norme universelle qui a pour objectif principal la spécification de données utilisées dans les protocoles de communication.
ATR	**Answer To Reset**. Suite d'octets envoyée par la carte lors de sa mise sous tension.
AVISPA	**Automated Validation of Internet Security Protocols and Applications**. Avispa est un outil pour la vérification et la validation formelle des protocoles de sécurité.
CA	**Certification Authority**. L'autorité de certification définit les politiques de gestion, de stockage et de vérification des certificats dans une une architecture PKI.
CAD	**Card Acceptance Device**. Périphérique qui est utilisé pour communiquer avec une carte à puce (lecteur de cartes).
CAP	**Convert APplet**. Le type de fichier standard pour les applications Java Card prêtes à être embarquées.
CRL	**Certificates Revocation List**. Liste de certificats révoqués dans une PKI.
EEPROM	**Electrical Erasable Programmable Read Only Memory**. Mémoire persistante (non volatile) dont le contenu est modifiable.

FRAM	**Ferroelectric Random Acces Memory**. Mémoire persistante (non volatile) dont le contenu est modifiable. Par rapport aux EEPROM, la FRAM est caractérisée par un nombre de cycle d'écriture plus grand et un accès plus rapide.
GlobalPlatform	Standard qui permet de gérer certains aspects realatifs à la sécurité dans les cartes multi-applicatives.
GSM	**Global System for Mobile communication**. Système mondial de communications mobiles pour le téléphone cellulaire.
HLPSL	**High Level Protocol Specification Language**. Langage de description de protocoles de sécurité utilisé par l'outil AVISPA.
JCRE	**Java Card Runtime Environment**. Environnement d'exécution pour les applets Java Card embarquées sur carte à puce.
JCVM	**Java Card Virtual Machine**. Machine virtuelle Java Card.
JVM	**Java Virtual Machine**. Machine virtuelle Java
MAC	**Message Authentication Code**. Code d'authentification de message qui sert pour la vérification de l'intégrité des données.
OSCP	**Online Certificate Status Protocol**. Protocole pour la validation en ligne d'un certificat.
PC/SC	**Personal Computer/Smart Card**. Standard de communication entre un PC et une carte à puce.
PKI	**Public Key Infrastructure**. Infrastructure à clé publique.
SCP	**Secure Channel Protocol**. Spécifie le canal de communictaion sécurisé établi avec une carte à puce.
SCVP	**Simple Certificate Validation Protocol**. Protocole pour la validation en ligne de certificats.
SIM	**Subscriber Identity Module**. Puce qui permet à l'utilisateur d'accéder au réseau de téléphone cellulaire.

Bibliographie

[1] M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *Proceedings of 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Paris, France, 2001.

[2] Ross Anderson. *Security Engineering : A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[3] Ross Jhon Anderson. Tamperproofing of Chip Card, 2000.

[4] Alessandro Armando and Luca Compagna. ATMC : a SAT-based model checker for security protocols. In *the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, Lisbon, Portugal, 2004.

[5] T. Autret, L. Bellefin, and M.-L. Oble-Laffaire. *Sécuriser ses échanges électroniques avec une PKI*. Eyrolles, 2002.

[6] Y. Avenel. Les OS pour cartes à puce entre ouverture et diversification. *Electronique Journal*, (136), 2005.

[7] *The AVISPA project*. `http://www.avispa-project.org/`.

[8] Faten Baccar and Monia Ben Brahim. Développement d'un orchestrateur de services sur JCG. Pojet de Fin d'Etude, École National d'Ingenieurs de Sfax, 2006.

[9] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. In *Workshop on Fault Detection and Tolerance in Cryptography*, Florence, Italy, 2004.

[10] Zeitcontrol. *BasicCard*. `http://www.basiccard.com/`.

[11] David Basin, Sebastian Mdersheim, and Luca Vigano. OFMC : A Symbolic Model-Checker for Security Protocols. *International Journal of Information Security*, 4, 2005.

[12] Lejla Batina, Elke De Mulder, Kerstin Lemke, Stefan Mangard, Elisabeth Oswald, and Gilles Piret. *Electromagnetic Analysis and Fault Attacks : State of the Art*. Information Societies Technology (IST) Programme, 2005.

[13] Fran Berman, Anthony J.G. Hey, and Geoffrey Fox. *Grid Computing : Making The Global Infrastructure a Reality*. John Wiley and Sons, 2003.

[14] Debojyoti Bhattacharya, Nitin Bansal, Amitava Banerjee, and Dipanwita RoyChowdhury. A near optimal s-box design. In *The 3rd International Conference of Information Systems Security, ICISS 2007*, Delhi, India, 2007.

[15] Christophe Bidan and Pierre Girard. La sécurité des cartes à microprocesseur. *Revue de l'électricité et de l'électronique*, (5), 2001.

[16] Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming. `http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf`, April 2001.

[17] Matt Bishop. *Computer Security : Art and Science*. Addison-Wesley, 2002.

[18] Y. Boichut, P.C. Heam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In *Third International Workshop on Automated Verification of Infinite-State Systems (AVIS'04)*, Barcelona, Spain, 2004.

[19] Vincent Bontems. L'art au temps des fractales. *Revue de synthèse*, 122, 2001.

[20] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. *Web Services Architecture. World Wide Web Consortium (W3C)*, February 2004. `http://www.w3.org/TR/ws-arch/`.

[21] Monia Ben Brahim. Tolérance aux pannes et prévention des erreurs dans une grille de cartes Java. Master's thesis, École Nationale d'Ingenieurs de Sfax, ENIS, 2007.

[22] Monia Ben Brahim, Faten Baccar, Achraf Karray, Maher Ben Jemaa, and Mohamed Jmaiel. Approche basée composition pour les applications sur une grille de cartes java. In *Septièmes Journées Scientifiques des Jeunes Chercheurs en Génie Electrique et Informatique, GEI 2007*, Monastir, Tunisise, 2007.

[23] C. Peltz. Web Services Orchestration and Choreography. *Journal of Computer*, 36(10), 2003.

[24] Ludovic Casset. *Construction Correcte de Logiciels pour Carte à Puce*. PhD thesis, Université d'Aix-Marseille II, October 2002.

[25] *CCID free software driver*. `http://pcsclite.alioth.debian.org/ccid.html`.

[26] *Certification Reports*. `http://www.bsi.de/zertifiz/zert/reporte.htm`.

[27] Alvin Chan, Florine Tse, Jiannong Cao, and Hong Va Leong. Enabling Distributed Corba Access to Smart Card Applications. *IEEE Internet Computing Journal*, 6(3), 2002.

[28] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. Resource management in Legion. In *the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, San Juan, Puerto Rico, 1999.

[29] Stéphanie Chaptal. Les cartes à puce géantes débarquent dans les mobiles. *01net Journal*, 2005.

[30] Serge Chaumette, Pascal Grange, Achraf Karray, Damien Sauveron, and Pierre Vignéras. Secure distributed computing on a Java Card grid. In *7th International Workshop on Java for Parallel and Distributed Computing*, Denver, Colorado, 2005.

[31] Serge Chaumette, Achraf Karray, and Damien Sauveron. Extended Secure Memory for a Java Card in the Context of the Java Card Grid project. In *11th IEEE Nordic Workshop on Secure IT-systems, NordSec 2006*, Linköping, Suède, 2006.

[32] Serge Chaumette, Achraf Karray, and Damien Sauveron. Gestion de la Sécurité pour l'Extraction Parallèle Distribuée des Connaissances. In *6èmes Journées Francophones Extraction et Gestion de Connaissances, EGC 2006*, Lille, France, 2006.

[33] Serge Chaumette, Achraf Karray, and Damien Sauveron. Secure Collaborative and Distributed Services in the Java Card Grid Platform. In *Workshop on Collaboration and Security (COLSEC 06)*, Nevada, USA, 2006.

[34] Serge Chaumette, Achraf Karray, and Damien Sauveron. The Software Infrastructure of a Java Card Based Security Platform for Distributed Applications. In *4th International Workshop on Security In Information Systems (WOSIS 2006)*, Phaos, Cyprus, 2006.

[35] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison Wesley, 2000.

[36] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Modersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In *Workshop on Specification and Automated Processing of Security Requirements (SAPS 2004)*, Linz, Austria, 2004.

[37] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *10th IEEE International Symposium on High-Performance Distributed Computing (HPDC10)*, 2001.

[38] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science Journal*, 1459, 1998.

[39] Michael L. Davis. Contactless smart cards : dangerous ways to use them. *Card Technology Today Journal*, 19, 2007.

[40] *DCSSI présentation*. `http://www.ssi.gouv.fr/fr/dcssi/index.html`.

[41] *Le grid-computing au service de la génomique et la protéomique*. `http://www.decrypthon.com/`.

[42] *he Community Resource for Jini Technology*. `http://www.jini.org/`.

[43] Server-based Certificate Validation Protocol (SCVP). `http://tools.ietf.org/html/draft-ietf-pkix-scvp-33`. (Draft Document).

[44] Olivier Dubuisson. *ASN.1 Communication entre systèmes hétérogènes*. Springer-Verlag, 1999.

[45] Christian Goire (*President of the Java Card Forum*). `http://www.javacardforum.org/openday_japan/1_introduction_president.pdf`, October 2007. presented in Japan Java Card Technology Forum.

[46] *EMVCo Website*. `http://www.emvco.com/`.

[47] EMV - Europay International S.A., MasterCard International Incorporated, and Visa International Service Association. *Integrated Circuit Card Specification for Payment Systems*, 1996.

[48] Zhimin Chen et Yujie Zhou. Dual-rail random switching logic : A countermeasure to reduce side channel leakage. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006)*, Yokohama, Japan, 2006.

[49] ETSI - European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+) ; Specification of the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface (GSM 11.11)*, 1995.

[50] *ETSI SIM*. `http://www.etsi.org/WebSite/Technologies/SIM.aspx`.

[51] Directive 95/46/ce du parlement européen et du conseil. *Protection des données à caractère personnel*. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:FR:HTML`.

[52] European Commission. Trans-European Access to Health Services for Mobile citizens. The NETC@RDS Project Website `http://netcards-project.com/web/frontpage`.

[53] Atallah Eve, Serge Chaumette, Franck Darrigade, Karray Achraf, and Damien Sauveron. A Grid of Java Cards to Deal with Security Demanding Application Domains. In *6th edition e-Smart conference & demos*, Sophia Antipolis, Frensh Riviera, 2005. *e-smart 2005 Isabelle Attali Award for the best innovative technology.*

[54] *FIPS 140-2* Security Requirements For Cryptographic Modules. `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`.

[55] Ian Foster. Globus Toolkit Version 4 : Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing (ICCS 2006)*, University of Reading, UK, 2006.

[56] Ian Foster and Carl Kesselman. Globus : A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2), 1997.

[57] Ian Foster and Carl Kesselman. The Globus project : a status report. *Future Generation Computer Systems Journal*, 15(5-6), 1999.

[58] Ian Foster and Carl Kesselman. *The Grid Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[59] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *The 5th ACM Conference on Computer and Communications Security*, 1998.

[60] *Smart Technology at Gemplus.*
`http://www.gemplus.com/smart/cards/basics/what.html`.

[61] Pierre Girard and Jean-Luc Giraud. Software attacks on smart cards. *Information Security*, 8(1), 2003.

[62] C. Giraud and H. Thiebeauld. A survey on fault attacks. In *Proceedings of CARDIS'04, Smart Card Research and Advanced Applications VI*, Toulouse, France, 2004.

[63] Global Grid Forum. *The Open Grid Services Architecture, Version 1.0.*
`http://www.gridforum.org/documents/GFD.30.pdf`.

[64] GlobalPlatform. *OpenPlatform Card Specification version 2.0.1.* GlobalPlatform Advancing Standards For smart Card Growth, 2000.

[65] GlobalPlatform. *Card Specification version 2.1.1.* GlobalPlatform Advancing Standards For smart Card Growth, 2003.

[66] GlobalPlatform. *Card Specification version 2.2.* GlobalPlatform The Standard For Card Infrastructure, 2006.

[67] *GlobalPlatform Card Specifications.*
`http://www.globalplatform.org/specificationview.asp?id=card`.

[68] *GlobalPlatform Device Specifications.*
`http://www.globalplatform.org/specificationview.asp?id=device`.

[69] *GlobalPlatform Systems Specifications.*
`http://www.globalplatform.org/specificationview.asp?id=system`.

[70] *GlobalPlatform website.*
`http://www.globalplatform.org/`.

[71] *The Globus Alliance.*
`http://www.globus.org/`.

[72] Li Gong. Secure java class loading. *IEEE Internet Computing*, pages 56 – 61, novemer - december 1998.

[73] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis (The "Duplication" Method). In *Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, Worcester, Massachusetts, USA, 1999.

[74] *Sun Grid Engine.*
`http://www.sun.com/software/gridware/`.

[75] Andrew Grimshaw and Wm Wulf. Legion : The next logical step toward the world-wide virtual computer. *Communications of the ACM Journal*, 40(1), 1997.

[76] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds. Legion : The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Departement of Computer Science, University of Virginia, Juin 1994.

[77] Object Management Group. The omg's corba website. `http://www.corba.org`.

[78] Groupe d'Intérêt Economique (GIE) Sesam-Vitale. Carte Vitale.
`http://www.sesam-vitale.fr/index.asp`.

[79] Patrick Gueulle. *Cartes à puce.* Editions Techniques et Scientifiques Françaises, 1993.

[80] Patrick Gueulle. *PC et cartes à puce.* Editions Techniques et Scientifiques Françaises, 1995.

[81] M.A. Hasan. Power Analysis Attacks and Algorithmic Approaches to Their Countermeasures for Koblitz Curve Cryptosystems. *IEEE Transactions On Computers Journal*, 50(10), 2001.

[82] Brant Hashii, Manoj Lal, Raju Pandey, and Steven Samorodin. Securing systems against external programs. *IEEE Journal on Internet Computing*, 2(6), 1998.

[83] Mike Henry. *Smart Card Security and Applications.* Artech House, 2001.

[84] *ISO - Internationnal Organization for Standardization.*
`http://www.iso.ch/`.

[85] *ISOIEC 15408.*
`http://standards.iso.org/ittf/PubliclyAvailableStandards/c040614_ISO_IEC_15408-3_2005(E).zip`.

[86] *ISO/IEC 17799 :2005.*
`http://www.iso.org/iso/fr/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39612`.

[87] H. Li R. Mullins J. A. Fournier, S. Moore and G. Taylor. Security evaluation of asynchronous circuits. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, Cologne, Germany, 2003.

[88] *Java Wrappers for PC/SC.*
`http://www.musclecard.com/middleware/files/jpcsc-0.8.0-src.zip`.

[89] *Java Card Technology.* `http://java.sun.com/javacard/3.0/`.

[90] *Java Card Technology.*
`http://java.sun.com/products/javacard/`.

[91] Sébastien Jean. *Modèles et Architectures d'Interaction interne et externe pour Cartes à Microprocesseur Ouvertes.* PhD thesis, Université de Lille I, Décembre 2001.

[92] Sébastien Jean and Didier Donsez. Extension des capacités de traitement des cartes à puce bases de données. Publication Interne au Laboratoire d'Informatique Fondamentale de Lille, 1999.

[93] Keith Jeffery. *Knowledge, Information and Data.* Council for the Central Laboratory of the Research Councils (CLRC), 1999.

[94] Default policy implementation and policy file syntax.
`http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html`.

[95] Roger Kehr, Michael Rohs, and Harald Vogt. Mobile Code as an Enabling Technology for Service-oriented Smartcard Middleware. In *2nd Internationnal Symposium on Distributed Objects and Applications DOA'2000*, University of California, Irvine, September 2000.

[96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *The 16th Annual International Cryptology Conference*, California, USA, 1996.

[97] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *The 19th Annual International Cryptology Conference*, California, USA, 1999.

[98] Oliver Kommerling and Markus G Kuhn. Design Principles for TamperResistant Smartcard Processors. In *the USENIX Workshop on Smartcard Technology (Smartcard99)*, Chicago, USA, 1999.

[99] Kundan Kumar, Debdeep Mukhopadhyay, and Dipanwita RoyChowdhury. Design of a Differential Power Analysis Resistant Masked AES S-Box. In *Progress in Cryptology INDOCRYPT 2007*, Chennai, india, 2007.

[100] Xavier Lagrange, Philippe Godlewski, and Sami Tabbane. *Réseaux GSM-DCS : des principes à la norme.* Hermès science, 1999.

[101] *LaserCard Web Site.*
`http://www.lasercard.com/index.php`.

[102] Xavier Leroy. Exploiting type systems and static analyses for smart card security. In *the CASSIS International Workshop : Construction and Analysis of Safe, Secure and Interoperable Smart devices*, Marseille, France, 2004. (exposé invité).

[103] Sergio Loureiro, Refik Molva, and Yves Roudier. Mobile code security. In *4ème Ecole d'Informatique des Systèmes Parallèles et Répartis, ISYPAR 2000*, Toulouse, France, 2000.

[104] Kamel Mazouzi. *JACE : un environnement d'exécution distribué pour le calcul itératif asynchrone.* PhD thesis, Université de Franche-Comté, 2005.

[105] Paul McFedries. The cloud is the computer. `http://www.spectrum.ieee.org/aug08/6490`, August 2008.

[106] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[107] Michael Montgomery and Ksheerabdhi Krishna. Secure Object Sharing in Java Card. In *the USENIX Workshop on Smartcard Technology (Smartcard'99)*, Chicago, USA, 1999.

[108] S. Moore, R. Anderson, R. Mullins, G. Taylor, and J. Fournier. Balanced self-checking asynchronous logic for smart card applications. *The Microprocessors and Microsystems Journal*, 27(9), 2003.

[109] *MultOS Web Site*. `http://www.multos.com`.

[110] Anand Natrajan, Marty Humphrey, and Andrew Grimshaw. Grids : Harnessing geographically-separated resources in a multi-organisational context. In *15th Annual International Symposium on High Performance Computing Systems and Applications*, Windsor, UK, 2001.

[111] Amaury Nève, Denis Flandre, and Jean-Jacques Quisquater. Feasibility of smart cards in silicon-on-insulator (soi) technology. In *USENIX Workshop on Smartcard Technology*, Chicago, USA, 1999.

[112] *Welcome to OpenCard*. `http://www.opencard.org/`.

[113] E. Oswald and M. Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In *3rd International Workshop on Cryptograpic Hardware and Embedded Systems(CHES)*, Paris, France, 2001.

[114] K Papapanagiotou, K. Markantonakis, Q. Zhang, W.G. Sirett, and K. Mayes. On the performance of certificate revocation protocols based on a Java Card certificate client implementation. In *20th IFIP International Information Security Conference*, Chiba, Japan, 2005.

[115] *PC/SC Workgroup website*. `http://www.pcscworkgroup.com/`.

[116] *MUSCLE PCSC Lite website*. `http://alioth.debian.org/projects/pcsclite/`.

[117] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic : DPA-resistance without routing constraints. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)*, Edinburgh, Scotland, 2005.

[118] *ProActive - Programming, Composing, Deploying on the Grid*. `http://www-sop.inria.fr/oasis/proactive/`.

[119] Jean-Jacques Quisquater and David Samyde. Cryptanalyse par side channel. In *Atelier SEcurité des Communications sur Internet (SECI'02)*, Tunis, Tunisie, 2002.

[120] Omer F. Rana and Luc Moreau. Issues in building agent based computational Grids. In *The 3rd Workshop of the UK Special Interest Group on Multi-Agent Systems, UKMAS00*, Oxford, UK, 2000.

[121] Jim Rees and Peter Honeyman. Webcard : a java card web server. In *Proceedings of the fourth working conference on smart card research and advanced applications (CARDIS'2000)*, Bristol, UK, September 2000.

[122] *IETF RFC N 2560*. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol, OCSP. `http://www.ietf.org/rfc/rfc2560.txt`.

[123] *IETF RFC N3280*. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `http://www.ietf.org/rfc/rfc3280.txt`.

[124] *IETF RFC 3748*. Extensible Authentication Protocol (EAP). `http://www.ietf.org/rfc/rfc3748.txt`.

[125] *IETF RFC 4017*. Extensible Authentication Protocol (EAP) Method Requirements for Wireless LANs. `http://www.ietf.org/rfc/rfc4017.txt`.

[126] David De Roure, Nicholas Jennings, and Nigel Shadbolt. *Research Agenda for the Semantic Grid : A Future e-Science Infrastructure*. Department of Electronics and Computer Science, University of Southampton, 2001.

[127] *RSA Laboratories, RSA-155 is factored !* `http://www.rsa.com/rsalabs/node.asp?id=2098`.

[128] Aviel Rubin and Daniel Geer. Mobile code security. *IEEE journal on Internet Computing Journal*, 2(6), 1998.

[129] Luis F. G. Sarmenta. Protecting Programs from Hostile Environments : Encrypted. Computation, Obfuscation, and Other Techniques. Department of Electrical Engineering and Computer Science (draft paper), 1999.

[130] *Smartcard.NET 1.0.* `http://www.hiveminded.com/sc10.htm`.

[131] *SETI@home website.* `http://setiathome.berkeley.edu/`.

[132] Smart Card Alliance. *Identity and Smart Card Technology and Application Glossary.* `http://www.smartcardalliance.org/resources/pdf/Identity_and_Smart_Card_Technology_and_Application_Glossary.pdf`.

[133] Sun Microsystem. *Java Card 2.2 Runtime Environment (JCRE) Specification*, 2002. Remote Method Invocation Service, chapter 8, pages 53-68.

[134] Sun microsystems. *Application Programming Interface Java Card Platform, Version 2.2.1.* http ://java.sun.com/products/javacard/specs.html.

[135] Sun microsystems. *Runtime Environment Specification Java Card Platform, Version 2.2.1.* `http://java.sun.com/products/javacard/specs.html`.

[136] Sun microsystems. *Virtual Machine Specification Java Card Platform, Version 2.2.1.* `http://java.sun.com/products/javacard/specs.html`.

[137] The AVISPA Team. *AVISPA v1.0 User Manual*, June 2005. `http://www.avispa-project.org/`.

[138] The AVISPA Team. *HLPSL Tutorial A Beginner's Guide to Modelling and Analysing Internet Security Protocols*, June 2005. `http://www.avispa-project.org/`.

[139] *Direction Centrale de la Sécurité des Systèmes d'Information DCSSI. La validation d'un certificat de clé publique*, February 2002. *Document édité par la DCCSI à l'occasion du séminaire sur les infrastructures de gestion de clés des 7 et 8 février 2002 à l'Ecole Militaire.*

[140] *GlobalPlatform Project.* `http://sourceforge.net/projects/globalplatform/`.

[141] E. Trichina, D. De Seta, and L. Germani. Simplified adaptive multiplicative masking for AES. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, San Francisco, USA, 2002.

[142] Hélène Trézéguet. Les microcontrôleurs pour cartes à puce. *Electronique Journal*, (141), 2003.

[143] M. Turuani. *Sécurité des Protocoles Cryptographiques : Décidabilité et Complexité.* PhD thesis, Université Henri Poincaré, Nancy, 2003.

[144] *Unicore Distributed Computing and Data Resources website.* `http://www.unicore.org/`.

[145] Pascal Urien and Marc Loutrel. La carte à puce EAP, un passeport pour la sécurité des réseaux émergents Wi-Fi. In *5èmes Journées Réseaux, JRES2003*, Lille, France, 2003.

[146] U.S General Services Administration. *Common Access Card Pre-Issuance Tech Requirements v 4.1.2 2/9/05*, 2005. `http://www.smart.gov/iab/documents/SmartCardPreissuanceSpecification.pdf`.

[147] P. Wohed, W. Aalst, M.P. Van Der, M. Dumas, and A.H.M. Hofstede. Analysis of Web Services Composition Languages : The Case of BPEL4WS. In *The 22nd International Conference on Conceptual Modeling*, Chicago, USA, 2003.

[148] Bennet Yee. A sanctuary for mobile agents. Technical report, UC at SanDiego, Dept. of Computer Science and Engineering, April 1997.