MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université Mouloud MAMMERI de Tizi-Ouzou
Faculté de Génie Électrique et Informatique
Département d'informatique

MEMOIRE

pour l'obtention du diplôme de

Magister en Informatique

Option : Informatique Fondamentale

Présenté par :

HABET Mohammed-Said

SUJET :

Conception et développement d'un protocole de gestion des transactions dans une plateforme Java Card

Devant le jury composé de :

Mr **LALAM Mustapha** ; Professeur, UMMTO	*Président*
Mme **BOUZEFRANE Samia** ; Maître de conférences, CNAM	*Rapporteur*
Mr **AHMED-OUAMER Rachid** ; Maître de conférences, UMMTO	*Examinateur*
Mme **AMEUR Zohra** ; Maître de conférences, UMMTO	*Examinatrice*

Année : 2009

Remerciements

Mon directeur de recherche, Madame Samia BOUZEFRANE, Maître de conférences au Conservatoire National des Arts et Métiers à Paris, qui a su me diriger vers la bonne voie, avec patience et générosité ; je la remercie pour toute l'aide qu'elle m'a prodigué.

Je remercie Monsieur Mustapha LALAM, Professeur à l'Université Mouloud MAMMERI de Tizi-Ouzou, pour l'honneur qu'il me fait de présider le jury de ma soutenance, et pour avoir accepté d'examiner ce travail.

Je remercie Monsieur Rachid AHMED-OUAMER, Maître de conférences à l'Université Mouloud MAMMERI de Tizi-Ouzou, pour l'honneur qu'il me fait de faire partie du jury de ma soutenance et d'examiner ce travail.

Je remercie Madame Zohra AMEUR, Maître de conférences à l'Université Mouloud MAMMERI de Tizi-Ouzou, pour l'honneur qu'elle me fait de faire partie du jury de ma soutenance et d'examiner ce travail.

Je tiens également à remercier toutes les personnes qui ont contribué, de prés ou de loin, à l'aboutissement de ce travail.

Table des matières

Introduction générale

La carte à puce (ou carte à microprocesseur) est un composant informatique portable, qui connaît un essor important depuis quelques années et qui, de par ses caractéristiques économique et informatique, devrait continuer à évoluer dans les années à venir [17].

Cependant, les cartes à puce exécutent des services dans un contexte très sujet aux pannes et aux interruptions de services : la carte est très souvent déconnectée, et elle peut être retirée du terminal dans lequel on la connecte à tout moment. Il faut donc rendre les cartes à puce ainsi que leurs partenaires d'exécution tolérantes aux pannes.

Parmi les nouvelles applications des cartes à puce, il y en a qui tendent à manipuler des données partagées sur une même puce. Cette manipulation si elle est faite sans précaution peut altérer la cohérence des données. Il faut donc assurer un contrôle de concurrence (à travers la sérialisation) qui puisse garantir le maintient de la cohérence (ainsi que l'isolation des applications).

La tolérance aux pannes et le contrôle de concurrence constituent les deux volets de la gestion des transactions.

Ce mémoire est organisé selon le plan suivant. Il comprend trois chapitres.

Le premier chapitre est consacré aux notions générales sur les cartes à puce, ainsi que l'architecture physique de ces cartes, et aussi leurs protocoles de communication. Nous donnerons également un aperçu du support logiciel (système d'exploitation) des cartes à puce.

Le second chapitre présentera les détails de l'architecture de la technologie Java Card tout en donnant des informations sur l'installation et l'exécution d'une applet sur une carte à puce.

Le troisième chapitre traite de la gestion des transactions tant de point de vue théorique (modèle transactionnel) que de point de vue pratique (transactions en Java Card).

CHAPITRE 1

Les Cartes à puce

Chapitre Un :

Les cartes à puce

I.1 Bref historique :

La prolifération des cartes en plastique a débuté aux USA au début des années 1950. Le bas prix de la matière synthétique polyvinyl chloride a rendu possible de produire des cartes plastiques robustes et durables qui étaient plus souhaitables pour l'utilisation quotidienne que le papier et carton utilisés précédemment et qui ne résistent pas aux pressions mécaniques et aux effets climatiques [26].

Le progrès énorme de la microélectronique dans les années 1970 a rendu possible d'intégrer une mémoire de données et une logique de traitement dans une simple puce de silicon mesurant quelques millimètres carrés. L'idée des cartes à puces a été pensée par deux allemands : Jürgen Dethloff et Helmut Gröttrup en 1967 ; ils déposèrent un brevet en 1969. Presque simultanément, un japonais, Kunitaka Arimura, puis un américain de chez IBM, Paul Castrucci déposèrent chacun un brevet relatif à la carte à puce respectivement en 1970 et 1971. En 1974 un français, Roland Moreno dépose 47 brevets dans 11 pays concernant les cartes à mémoire. En 1977, un ingénieur de CII-Honeywell-Bull, Michel Ugon, dépose les premiers brevets de la carte à microprocesseur (ou micro-calculateur).

En 1979 la première carte à base de microcontrôleur a vu le jour. Elle a été fabriquée par Motorola pour Bull CP8. Elle possède une UC de type 6805 (micro-contrôleur 8 bits de Motorola) avec une PROM de 1 Ko.

Les premiers essais des cartes à puce ont eu lieu en France et en Allemagne dans les débuts des années 1980 ; utilisation des cartes à puces comme cartes de prépaiement téléphonique et cartes bancaires débit/crédit sûres.

Récemment , avec les avancées dans la technologie des puces et cryptographie moderne ; les cartes à puce sont devenues plus puissantes. Elles sont utilisées maintenant pour mémoriser l'argent électronique , pour mémoriser et sécuriser les rapports médicaux personnels , pour empêcher l'accès non autorisé aux transmissions par câble et satellitaires , et pour améliorer la sécurité téléphonique sans fil ; leurs champs d'application ne cesse d'accroître jour après jour.

Première carte à puce (1979) :

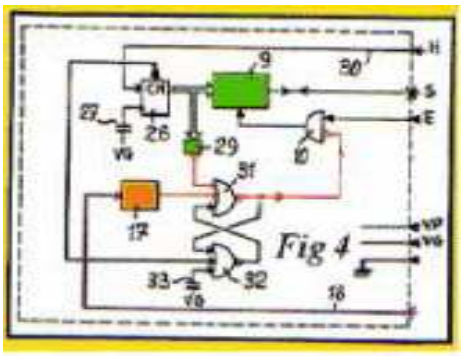


Figure I.a : CP8 : première carte à microcontrôleur

I.2 Cycle de vie de la carte :

● **Fabrication :**

- Inscription d'un programme en mémoire ROM définissant les fonctionnalités de base de la carte : «*masque*» figé sachant traiter un nombre limité de commandes pré-définies

● **Initialisation :**

- Inscription en EEPROM des données communes à l'application
- Possibilités pour certaines cartes d'ajouter des «*filtres*»

● **Personnalisation**

- Inscription en EEPROM des données relatives à chaque porteur

● **Utilisation**

- Envoi d'APDUs de commande à la carte
- Traitement de ces commandes par le masque de la carte
  - Si commande reconnue : - Traitement en interne de la commande ==> lecture/écriture de données en EEPROM
    - Renvoi d'un APDU de réponse
  - Si commande inconnue : Renvoi d'un code d'erreur

● **Mort**

- Par invalidation logique, saturation de la mémoire, bris, perte, vol, *etc.*

I.3 Types de cartes :

Les cartes à puce peuvent être classées sur différents axes : carte à contacts / carte sans contacts ; carte à mémoire / carte à microprocesseur.

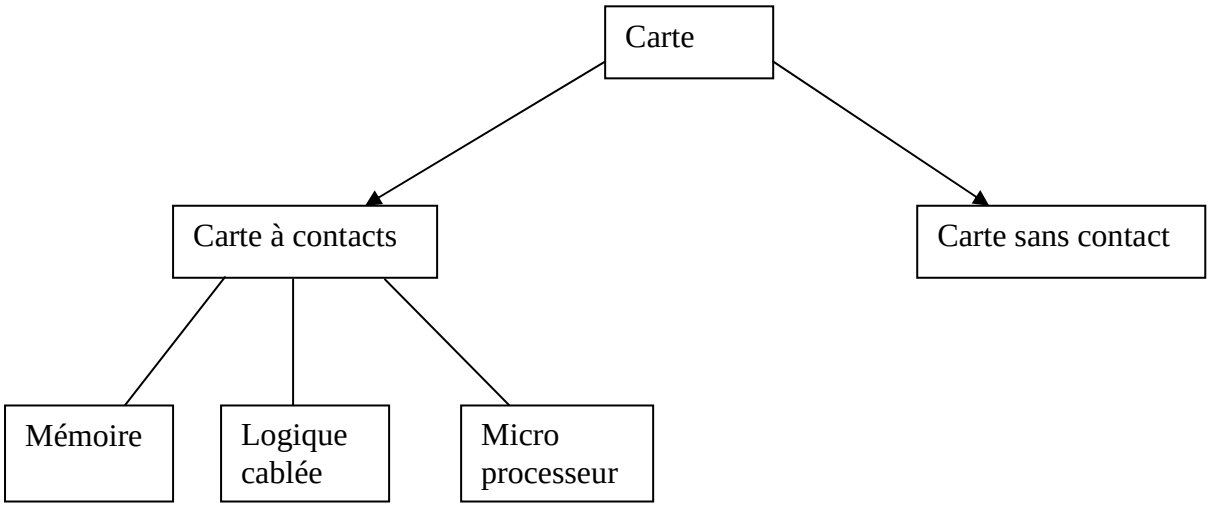


Figure I.b : classification des cartes

I.3.1 cartes à mémoire : [29]

Ce sont des cartes à mémoire simple (sans processeur) accessible en lecture sans protection, mais l'écriture peut être rendue impossible.

Les cartes mémoire OTPROM : (n'existent plus aujourd'hui)

exemple : télécarte de France Télécom

Les cartes mémoire OTPROM sont basées sur la technologie NMOS : haute tension (21 V) pour leur programmation (Vpp).

Elles sont dotées du circuit intégré ST 1200 (de SGS Thomson) : programmable une seule fois (mise à 1).

Elles sont d'une utilisation parfaite pour les télécartes (ne sont pas rechargeables). La sécurité repose sur la programmation une seule fois

Exemple de carte mémoire : carte téléphonique française 1ére génération

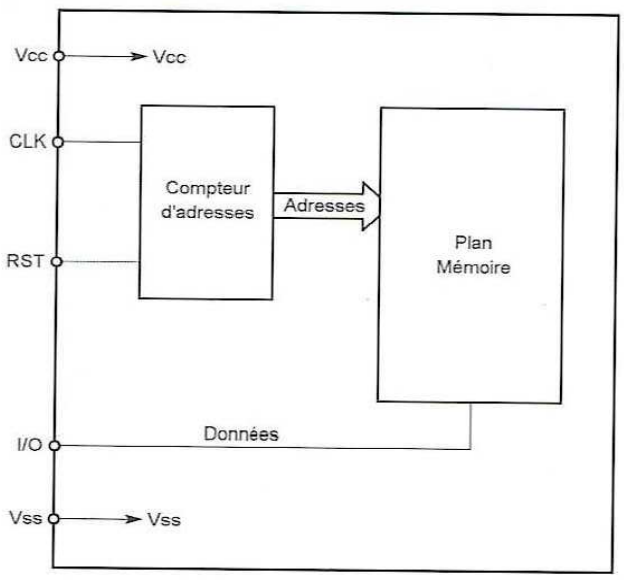


Figure I.c : Exemple de carte mémoire : carte téléphonique française 1ére génération

Synoptique interne d'une carte à mémoire OTPROM à base de ST 1200 :

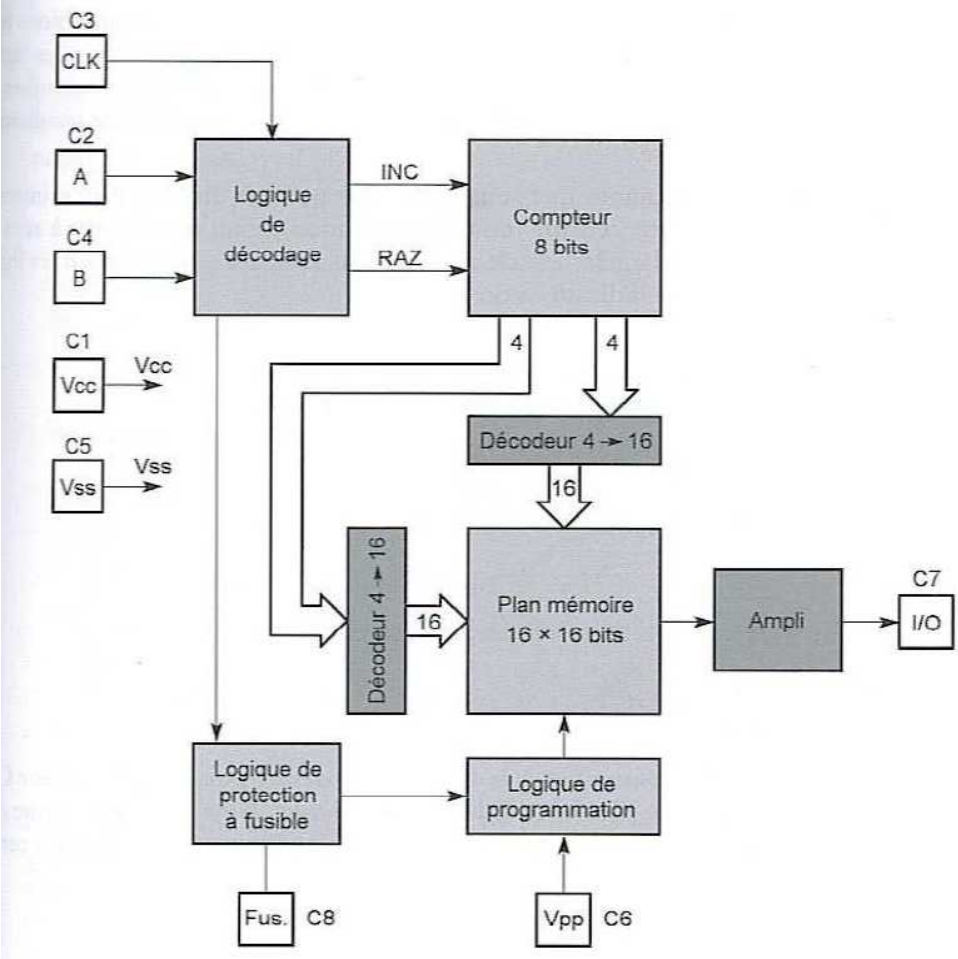


Figure I.d : Synoptique interne d'une carte à mémoire OTPROM à base de ST 1200

I.3.2 cartes à logique câblée :

La mémoire accessible via des circuits préprogrammés et figés pour une application particulière.

La carte comporte de la mémoire et des règles d'utilisation de celle-ci (certaines zones accessibles seulement en lecture). La logique des règles est implantée de manière physique dans le silicium du composant de la carte.

La carte peut effectuer des calculs figés : pas de souplesse, d'évolutivité de la carte : une caractéristique correspond à un ensemble de portes logiques.

Elle a conduit à la carte à microprocesseur.

Exemple 1 : Produit ATMEL référencé AT88SC1608

Exemple 2 : carte à logique câblée sécurisée

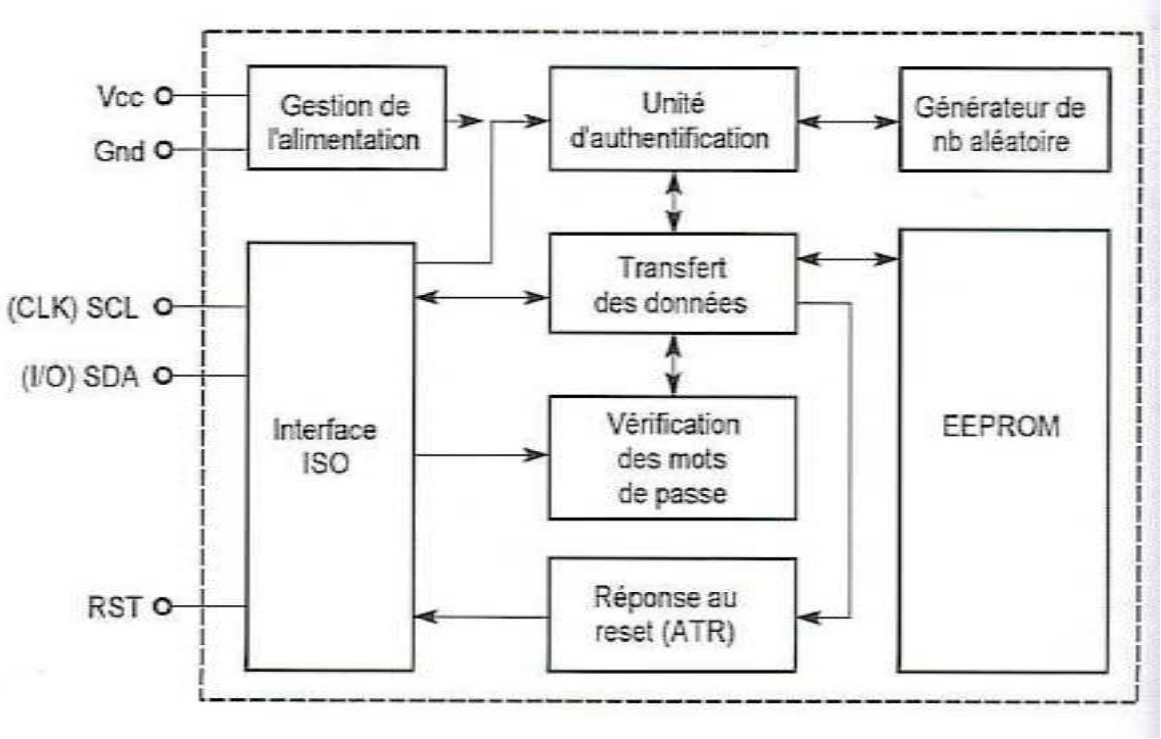


Figure I.e : carte à logique câblée

Développement d'une application :

On assimile la carte mémoire à X zones mémoires. Développer une application revient à programmer un certain nombre de registres internes de la carte.

Chaque zone contient :

- un octet : conditions d'accès à la zone (avec/sans mot de passe de Lect/Ecrit/, avec/sans authentification)
- données initiales pour l'authentification mutuelle (clé + semence du générateur de nbs aléatoires)
- mots de passe en Lect/Ecrit

I.3.3 cartes à microprocesseur : [29]

Ce sont des cartes où le microcontrôleur est encarté (processeur + mémoires).

La carte est programmable : elle peut effectuer tout type de traitements et peut intégrer plusieurs applications différentes. Ces cartes sont désignées aujourd'hui sous le vocable unique de carte à puce (ou *smart cards* en anglais). Les cartes à puce « intelligentes » renferment un microcontrôleur complet ; c'est-à-dire l'association en un seul circuit d'une unité centrale de microprocesseur, de mémoire morte, de mémoire vive, de mémoire EEPROM, d'une interface d'entrée/sortie série et de toute logique nécessaire pour faire fonctionner tout cela.

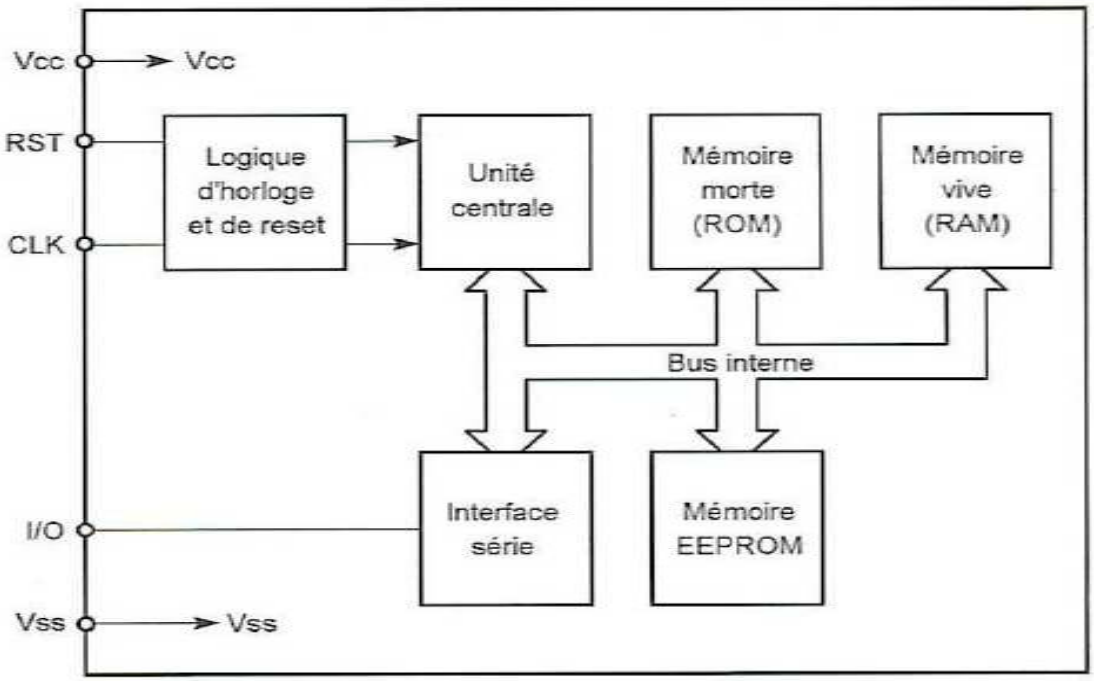


Figure I.f : Synoptique interne d'une carte à microcontrôleur

La première implémentation de la "smart card" (CP8) comportait :

RAM : 36 octets

EPROM : 1Ko

ROM : 1,6 Ko

Etapes de fabrication de la carte :

- Fabrication du support ou corps de la carte
- Plastique laminé
- Fabrication du composant (galette de silicium ou « wafer »)
- Fabrication du module :
- Découpage/Sciage,
- Contact/Binding,
- Protection dans le module,
- Collage.

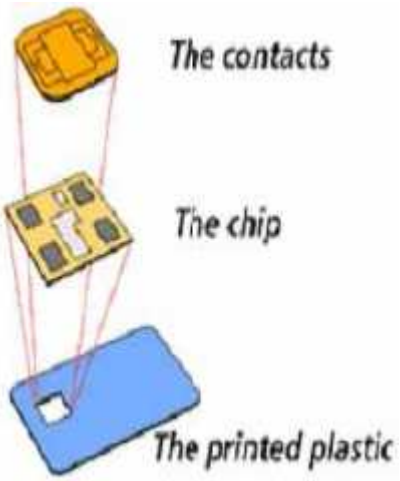
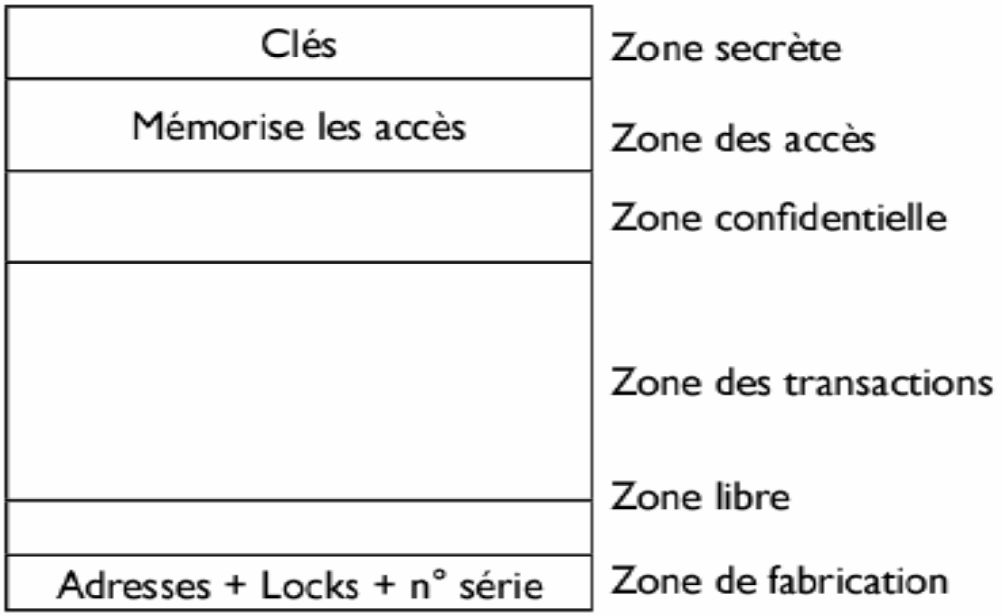


Figure I.g : mise en place de la puce dans la carte

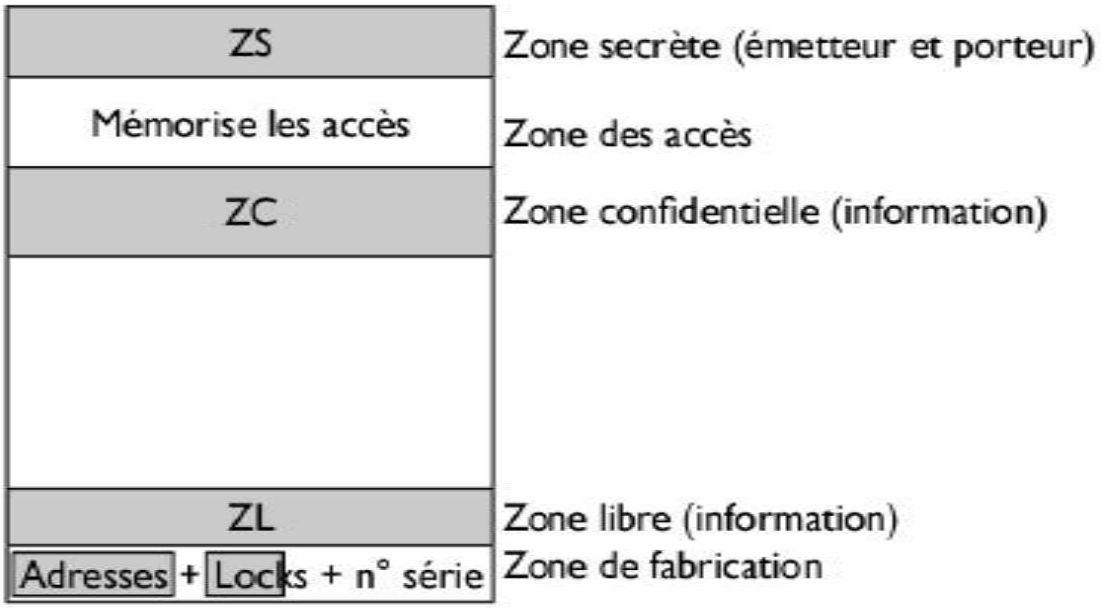
Exemple de « mapping » :



Fonctionnement :

A la pré-personnalisation (à la sortie de l'usine) : clé de fabrication

A la personnalisation :



A l'utilisation :

ZS	Zone secrète (émetteur et porteur)
Mémorise les accès	Zone des accès
ZC	Zone confidentielle (information) : *Lecture après présentation de code*
ZT	Zone transaction (information) : *Lecture & Écriture en fonction des protections*
ZL	Zone libre (information) : lecture possible
Adresses + Locks + n° série	Zone de fabrication

I.3.4 cartes sans contact :

Les données sont transférées sans contact électrique. Les cartes sans contact communiquent à travers une antenne (enroulée dans la carte) en utilisant le champ électromagnétique. Actuellement les cartes à mémoire et à microprocesseur sont toutes deux disponibles en cartes sans contact.

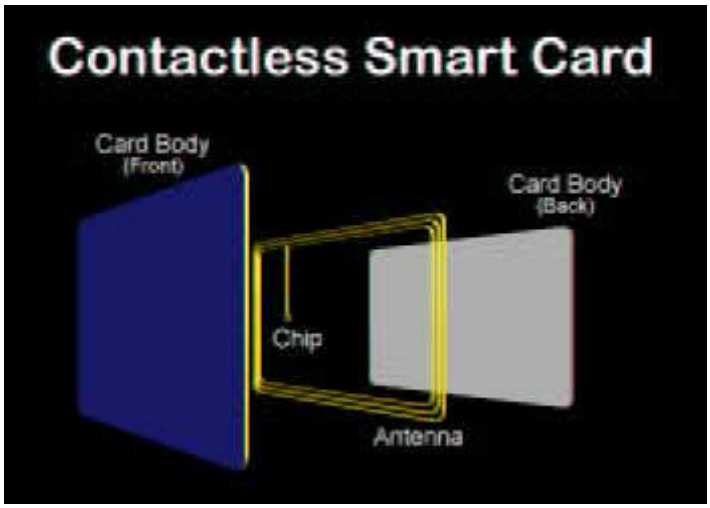


Figure I.h : Carte sans contact

I.4 Standardisation :

Une carte à puce est normalement un composant d'un système complexe. Cela signifie que les interfaces entre la carte et le reste du système doivent être précisément spécifiées et accordées les unes aux autres. En vue d'avoir des cartes indépendantes des applications et aussi des cartes multifonctionnelles , des standards ont été mis au point.

Les standards ISO/IEC :

Les standards ISO/IEC sont particulièrement significatifs puisqu'ils définissent les propriétés fondamentales des cartes à puce. ISO veut dire : International Organization for Standardization ; alors que IEC veut dire : International Electrotechnical Commission.

L'ISO est une organisation non nationale fondée en 1948. C'est une association mondiale de quelque 100 agences nationales de standards. Elle a pour tâche de promouvoir le développement de standards à travers le monde avec pour objectif de simplifier l'échange international de biens et services et développer la coopération dans les champs de la science et technologie et économie.

Les résultats des activités de l'ISO sont des accords publiés comme des standards ISO.

L'IEC est une organisation de standardisation dont le champ d'activité couvre les secteurs de la technologie électrique et l'électronique. Dans le but d'éviter la duplication de l'effort , les standards sont développés dans des comités techniques communs et sont publiés comme des rapports ISO/IEC.

Pour les cartes à puce le standard ISO 7816 « Idenfication cards – Integrated circuit cards with contacts » a été publié par l'ISO. C'est le plus important standard définissant les caractéristiques de la puce des cartes qui fonctionnent avec un contact électrique. Ce standard recouvre les sept aspects suivants des cartes à microprocesseur :

1. Aspect 1 : caractéristiques physiques de la carte ;
2. Aspect 2 : dimensionnement physique (extérieur) des contacts de la puce (brochage des contacts);
3. Aspect 3 : signaux électriques et protocoles de transmission (dialogue avec la carte) ;
4. Aspect 4 : commandes de base des cartes à puce (commandes inter industrie pour l'échange) ;
5. Aspect 5 : Identifiants d'applications ;
6. Aspect 6 : Eléments de données inter industrie ;
7. Aspect 7 : commande inter industrie pour SCQL.

I.5 Les caractéristiques physiques :

I.5.1 dimension :

La dimension typique d'une carte à puce est 85.6 mm (long) fois 54 mm (larg) fois 0.76mm (épaisseur).

Figure I.i : aspect et dimensions normalisées

exemple d'une carte à puce :

Figure I.j : exemple d'une carte à puce

I.5.2 les points de contact :

Les points de contact donnent accès aux éléments de la carte. Il existe huit points de contact. L'empattement et les critères physiques de ces points sont spécifiés dans l'ISO 7816-2.

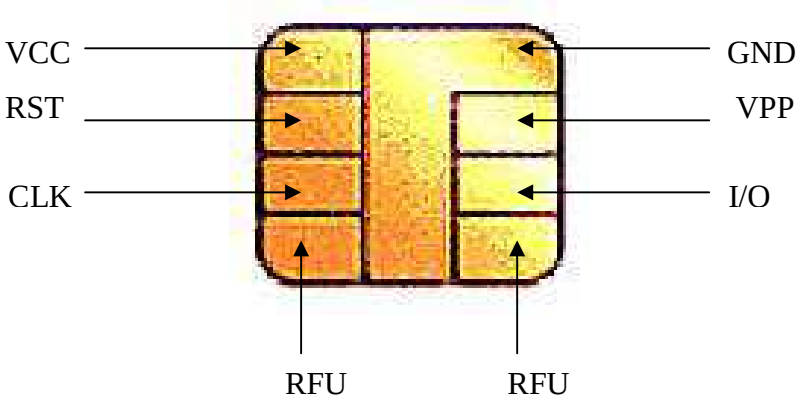


Figure I.k : brochage des contacts d'une carte à puce

Vcc : ce point apporte la source d'énergie qui permet le fonctionnement de la puce. Cette tension est comprise entre 3 et 5 volts (± 10%).

RST : cette entrée permet d'initialiser le microprocesseur , cette opération est appelée « warm reset » il existe également le « cold reset » qui correspond à une initialisation résultant du fait d'avoir coupé l'alimentation de la carte et de la rétablir ensuite.

CLK : il permet de fournir à la carte un signal d'horloge , la carte ne disposant pas de signal d'horloge Interne.

GND : il correspond à la masse, c'est le référentiel de tension.

Vpp : ce point est optionnel, il est utilisé par les anciens modèles de cartes qui ont besoin de deux sources d'alimentation de niveaux différents.

I/O : il est utilisé pour le transfert des données et des commandes entre la carte et le lecteur. La communication est de type half-duplex.

RFU : ces points sont réservés pour des utilisations futures.

I.5.3 Les Processeurs :

– Processeur central : cette unité est en général un micro contrôleur 8 bits avec une cadence d'horloge qui peut atteindre 5 MHz, en utilisant des multiplieurs de cadence on peut atteindre jusqu'à 40 MHz. Ils utilisent usuellement le jeu d'instructions de Motorola 6805 ou Intel 8051. Les nouvelles cartes à puce ont des microcontrôleurs de 16 bits ou de 32 bits ; et les cartes à puce dont le microcontrôleur est basé sur une architecture de type RISC sont également disponibles.

– Coprocesseur : c'est un processeur qui est spécialisé dans une tache, on trouve généralement des coprocesseurs cryptographiques qui sont conçus pour réaliser les calculs liés aux algorithmes de chiffrement tel que RSA.

I.5.4 Les mémoires :

Il existe différents types de mémoires qui ont chacune des contraintes particulières. Nous ne mentionnerons pas toutes les variantes de mémoires existantes mais juste un exemple par grande famille de mémoire :

– ROM (Read Only Memory) : comme son nom l'indique on ne peut écrire qu'une seule fois. Les données sont stockées de façon définitive à l'intérieur de ce type mémoire et elles sont conservées même après une coupure d'alimentation. Elle contient les routines du système d'exploitation ainsi que les données permanentes et applications utilisateurs. De plus, la lecture est le seul droit d'accès que l'on a sur ce type de mémoire.

– EEPROM (Electrical Erasable Programmable Read Only Memory) : les données sont modifiables électriquement, ce qui permet d'avoir des accès en lecture et en écriture à ce modèle de mémoire. D'autre part, tout comme les mémoires de type ROMs, les EEPROMs conservent leurs données même lorsque la carte est hors tension.

– RAM (Random Access Memory) : c'est un espace de travail temporaire car les données y sont conservées jusqu'à ce la carte soit hors tension. On peut avoir des accès en écriture et en lecture à ce type d'élément.

Il est important de préciser que les temps d'accès entre ces mémoires n'est pas le même . Le temps d'accès en écriture d'une EEPROM est 1000 fois plus lent que celui à une mémoire RAM.

Ce qui suit est la photographie de l'exemple d'une puce : la PC 83C852 avec les composants fonctionnels (du haut gauche à bas droite) : ROM , EEPROM , processeur , coprocesseur , et RAM. Cette puce a une superficie de 22.3 mm^2 et contient 1830000 transistors.

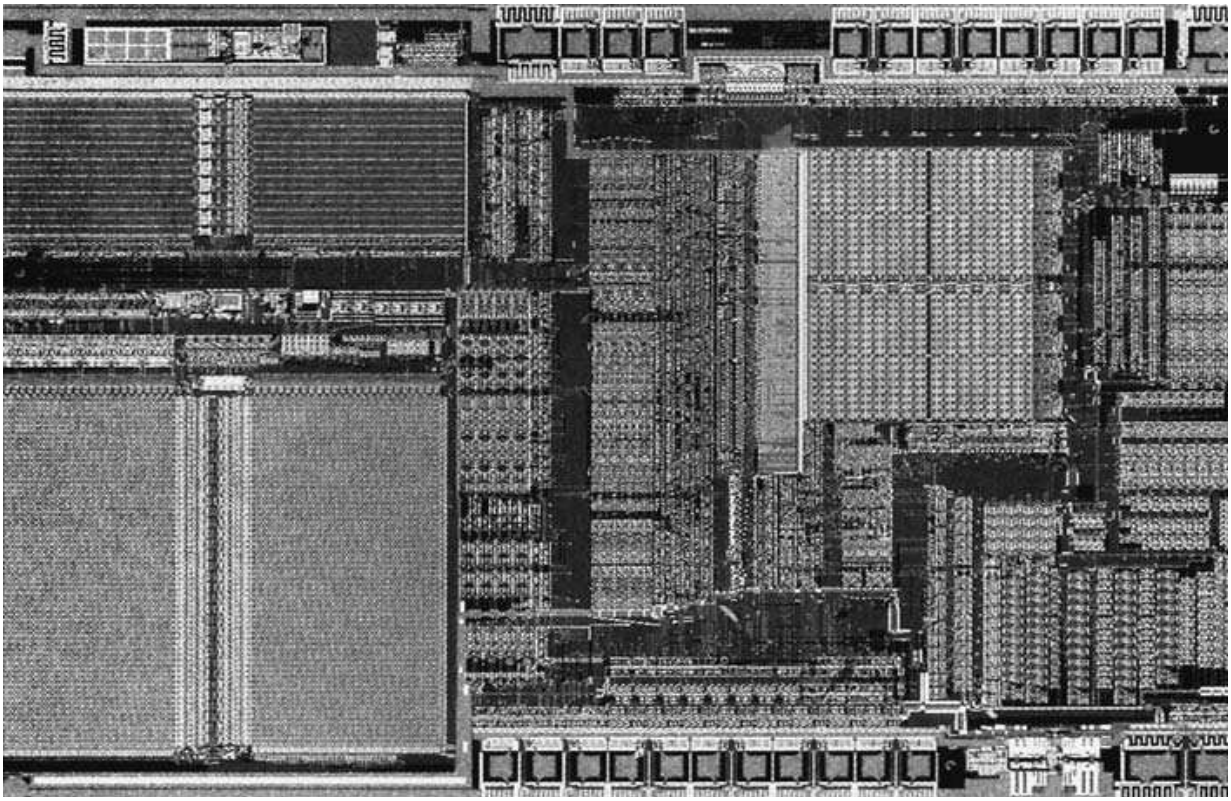


Figure I.l : exemple d'une puce : la PC 83C852

I.6 Communication :

I.6.1 modèle :

Le chemin de communication entre la carte et le hôte est semi-duplex : les données sont soit transmises du hôte vers la carte ou alors de la carte vers le hôte mais non les deux en même temps.

Quand deux ordinateurs communiquent, ils échangent des paquets de données , qui sont construits suivant un protocole , tel que TCP/IP. De la même façon les cartes à puce dialoguent avec les autres ordinateurs en utilisant leurs propres paquets – appelés APDU (application protocol data unit).

Une APDU contient une commande ou un message de réponse.

Dans le monde des cartes , le modèle maître-esclave est utilisé. La carte à puce joue toujours le rôle passif (esclave) attendant une commande APDU de l'hôte. Elle exécute ensuite l'instruction spécifiée dans la commande et répond à l'hôte avec une réponse APDU. Les APDU commandes et APDU réponses sont échangées alternativement entre la carte et le hôte comme le montre la figure suivante :

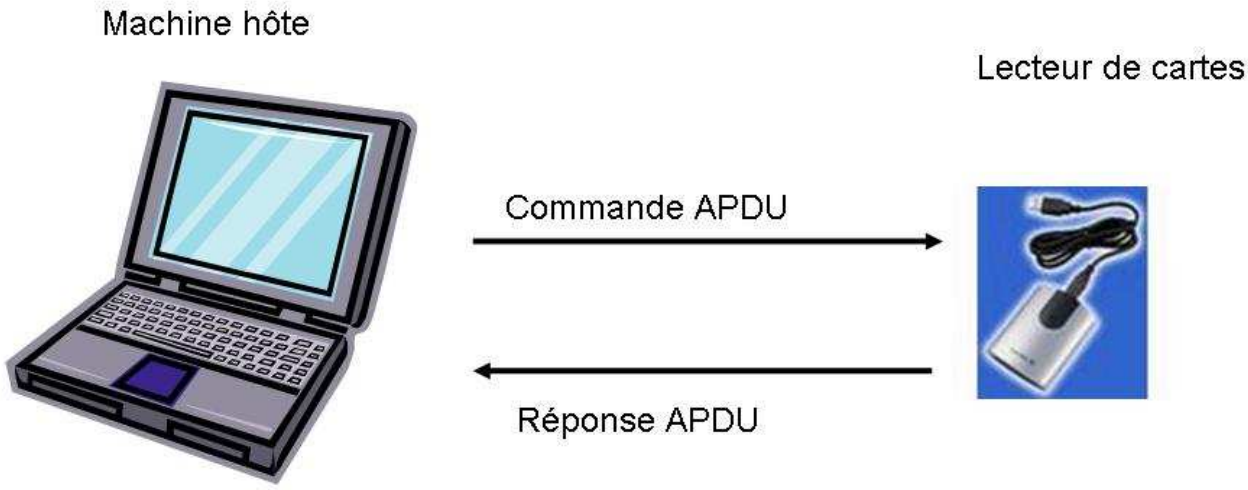


Figure I.m : le protocole APDU

I.6.2 le protocole APDU :

Le protocole APDU , comme spécifié dans l'ISO 7816-4 , est un protocole de la couche application entre la carte à puce et l'application hôte. Cette spécification distingue entre deux types de messages :

- C-APDU : (Command APDU)

Commande APDU						
Entête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Data field	Le

Entête (Mandatory header) :

• CLA (1 octet): Classe d'instructions , indique la structure et le format pour une catégorie de commandes et de réponses APDU ;

• INS (1 octet): code d'instruction, spécifie l'instruction de la commande ;

• P1 (1 octet) et P2 (1 octet): paramètres de l'instruction , fournissent des informations supplémentaires quant aux qualifications de l'instruction ;

Champs optionnels (Optional body) :

• Lc (1 octet): il indique le nombre d'octets présents dans le champ données « data Field » de la commande

• Data field (octets dont le nombre est égal à la valeur de Lc): une séquence d'octets dans le champ données de la commande représentant les données utiles transmises à la carte pour exécuter l'instruction spécifiée dans CLA.

• Le : il indique à la carte le nombre d'octet que l'application cliente attend en réponse.

- R-APDU : (Response APDU)

Réponse APDU		
Corps optionnel	Partie obligatoire	
Data field	SW1	SW2

Champs optionnels (Optional body) :

• Data Field, ce champ est une séquence d'octets contenant les données de réponse de la carte à l'application cliente. Sa taille doit correspondre à celle qui est attendue par le client en réponse et que ce dernier a spécifié dans son message C-APDU dans le champ Le.

Partie obligatoire (Mandatory trailer) :

• SW1 et SW2, ces éléments constituent le « status word » (mot d'état) , ils permettent de connaître l'état de la carte après l'exécution de la C-APDU.

SW1 SW2 = 0x90 0x00 Succès
0x6E 0x00 CLA error
0x6D 0x00 INS error
0x6B 0x00 P1, P2 error
0x67 0x00 LEN error
0x98 0x04 Bad PIN
0x98 0x40 Card blocked

Exemples de cartes :

Champ de la Commande APDU	Valeurs
CLA	BC = cartes de crédit françaises, cartes vitales françaises, A0 = cartes SIM (téléphonie) 00 = cartes Monéo (porte-monnaie en France), Mastercard, Visa
INS	20 = vérification du PIN, B0 = Lecture B2 = Lecture de record D0 = Écriture DC = Écriture de record A4 = Sélection du répertoire (directory) C0 = Demander une réponse (get an anwer)
P1, P2	paramètres contenant des adresses à lire
LEN	Longueur prévue pour la réponse ou bien longueur de l'argument de l'instruction
ARG	contient LEN octets (octets à écrire, PIN à vérifier, etc.)

Le champ « Data Field » est optionnel dans les deux types d'APDU (command et response). Il y a alors quatre configurations possibles de l'APDU qui peuvent être formatés en fonction du fait que le champ « Data Field » soit rempli ou pas :

1. Il n'y a pas de données à transférer en direction de la carte ou de l'application cliente, alors dans ce cas le message de type C-APDU contient uniquement une entête et le R-APDU ne contient que les informations relatives aux statuts.

• C-APDU :

CLA	INS	P1	P2

• R-APDU :

SW1	SW2

2. Il n'y a pas de donnée à transférer vers la carte mais des données sont attendues en retour. Le message contient l'entête et le corps de la commande. La partie « optional body » contient un octet (le champ Le) qui indique le nombre de donnée que l'on attend en retour.

• C-APDU :

CLA	INS	P1	P2	Le

• R-APDU :

Data field	SW1	SW2

3. Il y a des données à transférer vers la carte et aucune donnée n'est attendue en réponse. Le message de type command contient l'entête (Mandatory header) et les champs du corps optionnel contiennent les informations : Lc et « Data Field ». Le message de retour contient uniquement les données de statuts (trailer status).

• C-APDU :

CLA	INS	P1	P2	Lc	Data field

• R-APDU :

SW1	SW2

4. Il y a des données à transférer vers la carte et des données sont attendues en retour. Le message de type commande contient l'entête (Mandatory header) et les données des champs Lc, « Data Field » et « Le ». Le message de type réponse contient les donnés de retour « Data Field » et les informations de statuts.

• C-APDU :

CLA	INS	P1	P2	Lc	Data field	Le

• R-APDU :

Data field	SW1	SW2

I.6.3 le protocole TPDU :

Les APDU sont transmises par le protocole de niveau suivant – le protocole de transport , défini dans l'ISO 7816-3. Les structures de données échangées entre un hôte et une carte en utilisant le protocole de transport sont dites *Tansmission Protocole Data Units* ou TPDUs.

Les deux protocoles qui sont utilisés dans les systèmes de carte à puce sont le protocole T=0 et le protocole T=1. Le protocole T=0 est orienté octet , ce qui signifie que la plus petite unité traitée et transmise par le protocole est un simple octet. Par contraste le protocole T=1 est orienté bloc ; en d'autres termes le bloc , consistant en une séquence d'octets , est la plus petite unité de données qui peut être transmise entre une carte et un hôte.

I.6.4 ATR : (Answer To Reset)

Dès que la carte est mise sous tension, elle envoie un message de réponse d'initialisation appelé ATR, il peut atteindre une taille maximum de 33 octets. Il indique à l'application cliente les paramètres nécessaires pour établir une communication avec elle. Il fournit entre autre les informations suivantes :

– Le protocole de transport ;

– Taux de transmission des données ;

– Numéro de série de la puce …

Le comportement de la carte lors d'un Reset est illustré dans la figure suivante :

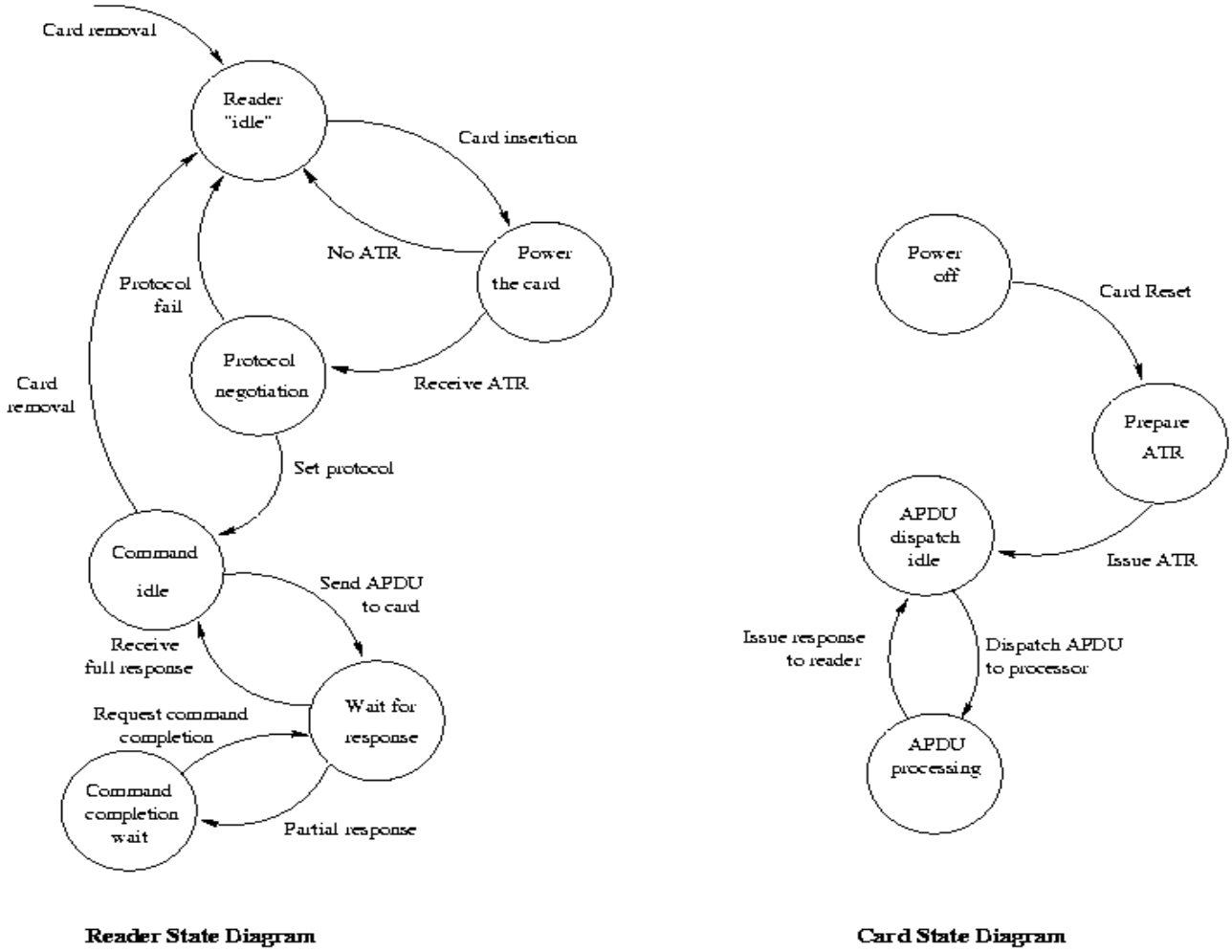


Figure I.n : comportement de la carte lors d'un Reset

I.7 Systèmes d'exploitation et cartes à puce :

Aujourd'hui les systèmes d'exploitation répondent à trois besoins dissociés :

1. **Les systèmes d'exploitation gèrent le matériel**. L'utilisation du matériel nécessite en général d'exécuter certains traitements logiciels qui n'ont rien à voir avec le but dans lequel on sollicite le matériel. Dans une carte à microprocesseur , par exemple, l'accès en écriture à une page mémoire Flash n'est pas une opération élémentaire du microprocesseur. Une routine dédiée charge « les pompes d'écriture ». Elle veille à respecter certains « délais de garde » durant l'effacement de la mémoire, puis elle écrit. De plus en cas d'écriture répétée la page Flash peut être « stressée » et devient alors inutilisable. Les systèmes d'exploitation gèrent le matériel de telle sorte que toutes ces considérations matérielles ne soient pas prises en charge par les applications encartées.

2. **Les systèmes d'exploitation fournissent une abstraction fonctionnelle du matériel.** La mémoire persistante des cartes est par exemple souvent représentée au programmeur sous la forme d'un système de fichiers. C'est le cas des normes ISO 7816-4 ou des *SmartCard for Windows*. Elle est parfois aussi présentée comme un support d'objets persistent (dans les cartes Java). L'objectif est de fournir au concept-

-eur d'applications une manière conviviale d'utiliser le support physique. Aucune règle particulière ne guide en fait le modèle d'abstraction qui est défini par le système d'exploitation. Il en va de même pour les entrées/sorties (qui dans la carte sont présentés sous forme d'APDU) et même du microprocesseur qui est maintenant masqué par la notion de machine virtuelle.

3. **Les systèmes d'exploitation protègent les ressources fournies aux différentes applications**. Cette tâche a pour but d'assurer la pérennité des applications. Lorsque plusieurs programmes coexistent sur un même support d'exécution , l'activité de l'un peut nuire au fonctionnement de l'autre. Dans une carte un programme qui gère un porte-monnaie électronique stocke en mémoire persistante le code secret qui permet son utilisation. Si cette ressource mémoire peut être atteinte (simplement lue en l'occurrence) par un autre programme chargé ultérieurement , alors ce programme nuit au fonctionnement de l'application porte-monnaie. Pour pallier à ce problème, le système d'exploitation est alors pensé comme un médiateur entre les différentes applications : il devient le noyau de confiance auquel les ressources accordées à chaque programme sont sécurisées.

Tous les systèmes d'exploitation répondent à ces trois critères. Chacun apporte des approches différentes qui amènent à des architectures différentes. Finalement, le critère qui les distingue les uns des autres porte sur la frontière qu'ils définissent avec les applications sur les trois axes que sont :
la gestion du matériel, l'abstraction du matériel et la sécurité des programmes. Cette frontière entre système et applications se manifeste le plus souvent par des pénalités qui augmentent le temps machine consommé et qui limitent l'utilisation du matériel. Ces pénalités sont généralement induites par certains mécanismes choisis par les systèmes. Mais il est aussi possible de refuser cette séparation. Tel est le cas des systèmes d'exploitation intégrés.

I.7.1 Systèmes d'exploitation intégrés :

Historiquement, avant que le concept de système d'exploitation ne soit clairement identifié, les programmes réalisaient indifféremment des tâches applicatives et des tâches de gestion du matériel.
Il était impossible de distinguer dans le programme ce qui relevait de la gestion du matériel, de ce qui relevait des traitements applicatifs. Au mieux quelques routines de gestion du matériel étaient systématiquement réutilisées. Dans ce cas les bibliothèques constituées par ces routines représentaient le système d'exploitation. De fait, on constate dans les systèmes intégrés que la fonction d'abstraction disparaît partiellement ou complètement. L'application conçue en même temps que le système, s'appuie directement sur les routines de gestion de matériel. L'une des conséquences est que la conception de nouveaux logiciels devient une tâche ardue, aussi le délai de conception est particulièrement important.
Cette approche, très ancienne dans les systèmes informatiques conventionnels a dès les années 1964-1965, été remplacée de véritables systèmes d'exploitation. Dans le contexte des cartes à puce, cette technique est longtemps restée la seule manière de faire. Elle s'est développée plus qu'ailleurs car elle permet d'améliorer de manière très sensible les performances des applications tout en diminuant d'autant la taille

totale des logiciels encartés, ce qui est essentiel pour réduire les coûts de production des fabricants. Le travail du programmeur est de trouver « le plus court chemin » entre les services accomplis par son logiciel et le matériel dont il dispose. Des systèmes formels et des techniques de test ont été spécialement développées pour prouver , avant leur émission , la viabilité des applications à systèmes intégrés qui étaient encartées. Jusqu'à la fin des années 1990, les spécialistes de la carte parlaient de système d'exploitation pour désigner le tout que formaient les logiciels encartés. Ce n'est qu'avec l'accroissement de la puissance du matériel disponible dans les cartes que les applications intégrées ont été remplacées progressivement par les premiers véritables systèmes d'exploitation monolithiques.

I.7.2 Systèmes d'exploitation monolithiques :

Les systèmes d'exploitation monolithiques sont pensés comme un tout. Dans le meilleur des cas ils gèrent le matériel, en proposent une abstraction et assurent la sécurité (en protégeant les ressources fournies aux applications). Ils constituent une couche logicielle placée entre le matériel et les applications.

Le système de fichiers est l'exemple le plus conventionnel d'abstraction que propose un système d'exploitation monolithique. La figure I.o présente l'architecture interne de la partie « gestion de fichier » (sur le principe de la norme ISO 7816-4) dans une carte à microprocesseur. La couche supérieure assure l'interface entre le système d'exploitation et les applications. Elle ne correspond pas au cœur du système d'exploitation. Cependant on distingue sur le graphique le système se subdivise lui-même en sous-couches. Au plus bas on trouve les routines de gestion du matériel. Elles gèrent l'allocation des unités physiques de mémoire et leurs accès. Sur cette base s'appuie la gestion du regroupement des unités physiques en unité logique et de l'accès aux unités logiques indépendamment les unes des autres.

Traditionnellement c'est sur les unités logiques qu'est placé un mécanisme de contrôle d'accès pour garantir la sécurité-innocuité es applications.

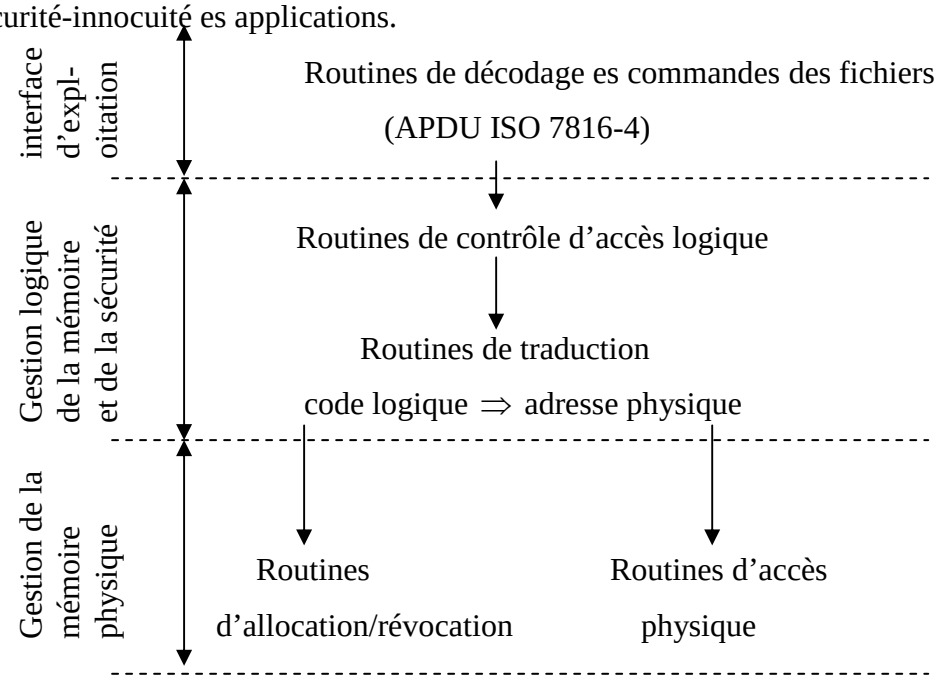


Figure I.o : Architecture d'un système de gestion de fichiers encarté

Les couches internes de la gestion d'un système monolithique ne doivent pas nous faire perdre de vue que le propre de ce type de système est de présenter la gestion du matériel au travers d'une abstraction unique. Les sous-couches sont bien évidemment inaccessibles aux applications, faute de quoi les mécanismes de sécurité pourraient être contournés. Le propre d'un système monolithique est d'être une « offre tout-en-un » : une gestion du matériel + une abstraction + un système de sécurité. Dans la pratique, cette intégration des différents aspects permet de réaliser un bloc fonctionnel plus performant que s'ils étaient dissociés. Toutefois, l'abstraction unique qu'ils proposent aux applications se montre parfois totalement inadaptée. Les applications orientées bases de données se trouvent par exemple fortement pénalisés lorsqu'elles sont réalisées « au dessus » de systèmes qui représente la mémoire persistante sous forme de fichiers. Finalement, on retrouve à un niveau moindre le même phénomène que celui des applications à systèmes intégrés : la complexité croissante des tâches assumées par le noyau des systèmes monolithiques les rend de plus en plus difficiles à maintenir ; elle les rend aussi de moins en moins réutilisables. Pour maîtriser cette complexité, la communauté des systèmes d'exploitation a cherché à réduire la taille du noyau, ce qui a abouti au concept de micro-noyau.

I.7.3 Micro-noyau :

L'objectif principal des micro-noyaux est de parvenir à maîtriser la complexité croissante des logiciels qui constituent les systèmes d'exploitation. Les concepteurs ont cherché à rendre plus synthétiques, plus homogènes les abstractions qui sont proposées. Aussi l'une des principales caractéristiques d'un micro-noyau est la taille réduite de son code. Comme ils sont plus petits que les systèmes monolithiques, ils sont aussi plus facilement compris et leur maintenance est simplifiée. La taille d'un micro-noyau peut aller d'un kilo octets et demi à une centaine de kilo octets. Malgré cette première caractéristique attrayante dans un contexte minimaliste, ces architectures n'ont jamais su retenir l'attention des concepteurs de systèmes à carte à microprocesseur. Mais la réduction de la taille du noyau d'un système a aussi pour conséquence de limiter le nombre de services proposés. En fait, cette approche montre que les programmeurs système renoncent à implanter toutes les abstractions que les applications peuvent réclamer.

Les micro-noyaux cherchent à proposer un découpage des ressources matérielles et une abstraction sécurisée plus simple et plus flexible que celle proposée par les systèmes monolithiques. Les applications peuvent ensuite réutiliser les interfaces du micro-noyau pour définir leurs propres modèles de mémoire persistante, par exemple.

La principale difficulté pour réaliser un micro-noyau est de réussir à proposer un modèle de base véritablement réutilisable et performant. Il existe aujourd'hui deux familles de micro-noyaux correspondant à des stratégies différentes.

La première stratégie consiste à proposer des mécanismes permettant d'étendre le contenu du micro-noyau. Ainsi, chaque application qui souhaite un service inconnu du noyau peut l'y ajouter.

Les performances des composants ajoutés sont supérieures à ce qu'elles seraient si elles étaient placées du coté de l'application car elles ne subissent plus les pénalités de performance lorsqu'elles sollicitent les interfaces de base du micro-noyau. La difficulté est alors de maintenir un niveau de sécurité suffisant alors que les problèmes peuvent venir du noyau du système lui-même. Les systèmes SPIN [33] et Vino [37] exposent les problèmes rencontrés dans l'extension du logiciel placé dans le noyau.

La seconde stratégie consiste à proposer des micro-noyaux qui exposent des interfaces d'exploitation plus proches du matériel géré [34],[36]. Ainsi l'implantation de nouveaux mécanismes n'est plus contrainte que par le matériel. Les routines de base proposées par le système d'exploitation ne se présentent plus comme une couche de traitements, parfois inutiles (voir néfastes), mais imposées de fait par les ressources physiques qui sont manipulées. Dans cette approche, la mémoire persistante, par exemple, n'est plus présentée comme un support de fichiers, mais comme un ensemble de pages avec une sémantique d'exploitation proche de celle des secteurs de disque manipulés pour les représenter. Libre alors à chacun d'exploiter ces pages pour implanter des fichiers, des objets Java ou des lignes d'une table CQL. Chacun peut exploiter différemment les ressources matérielles.

I.7.4 Exploiter différemment les ressources matérielles :

Finalement, aujourd'hui, deux domaines d'investigation scientifique différents se dessinent derrière la conception d'un système d'exploitation.

Un système d'exploitation, en tant que support d'exécution, doit pouvoir répondre aux exigences croissantes des applications. Il devient impératif de parvenir à faire la synthèse des besoins que les programmeurs manifestent vis-à-vis des plateformes qui exécutent leurs applications. Dans [13] (section 2.3) est présentée au travers d'une architecture novatrice comment il est possible de formuler une réponse cohérente aux attentes de plus en plus diversifiées des concepteurs d'application. Cependant un système d'exploitation gère avant tout des ressources physiques. Les reproches adressés aux systèmes existants portent presque toujours sur des limitations qui sont induites par l'abstraction du matériel qui est proposé par le système. C'est en partant de cette constatation [35] qu'une équipe de chercheurs du MIT a cherché à définir une nouvelle architecture de systèmes d'exploitation plus efficaces, fiables et extensibles qui se place au plus près du matériel : les exo-noyaux.

I.7.5 systèmes d'exploitation des cartes à puces :

Les systèmes d'exploitation des cartes à puce ont une légère ressemblance avec les systèmes d'exploitation de bureau tels que UNIX, Microsoft Windows , DOS. Les systèmes d'exploitation des cartes à puce sont conçues pour travailler avec l'interface série bidirectionnelle au terminal : ils supportent une collection d'instructions sur lesquelles les applications utilisateurs sont construites. L'ISO 7816-4 standardise un large champ d'instructions dans le format d'APDU. Un système d'exploitation de carte à

puce supporte quelques unes ou toutes ces APDUs tout comme les additions et les extensions du fabricant. La plus part des systèmes d'exploitation des cartes à puce supportent un modeste système de fichier basé sur l'ISO 7816-4. Les APDU 7816-4 sont largement des commandes orientées système de fichier telles que les commandes de sélection de fichier et accès au fichier.

I.7.5.1 systèmes de fichier des cartes à puce :

Les cartes à puce définies dans l'ISO 7816-4 peuvent avoir une structure de système de fichier hiérarchique comme indiqué dans la figure ci-dessous. Le système de fichier ISO 7816-4 supporte trois types de fichiers :

– MF, master file ;

– DF, dedicated file ;

– EF, elementary file.

L'identification de ces fichiers se fait par un identifiant de 2 octets ou un nom symbolique d'une taille maximum de 16 octets.

Avant qu'une opération soit réalisée, il faut que le fichier soit sélectionné. De plus, l'accès à ces fichiers peut être contrôlé en spécifiant des droits d'accès.

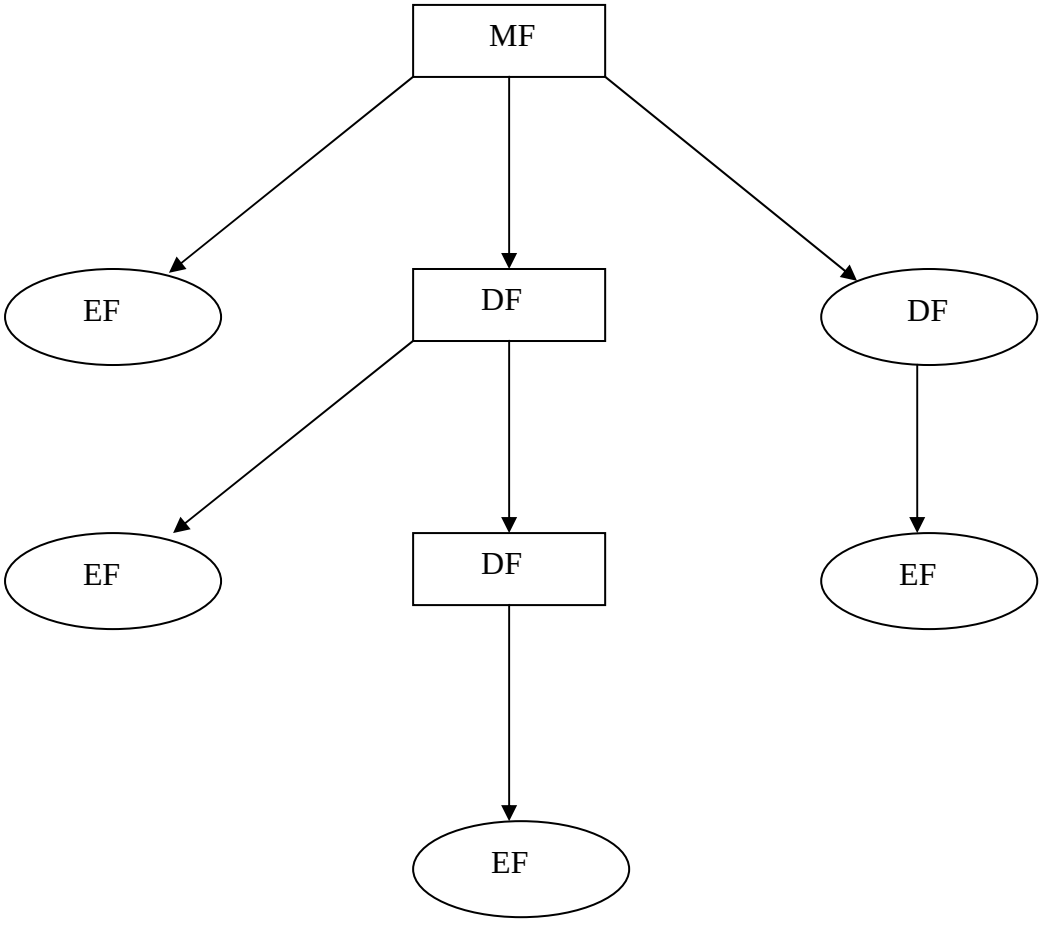


Figure I.p : structure du système de fichier ISO 7816-4

I.7.5.2 master file :

Le MF est la racine du système de fichier. Il peut contenir des « dedicated files » et « elementary files ». Il y a seulement un MF dans une carte à puce.

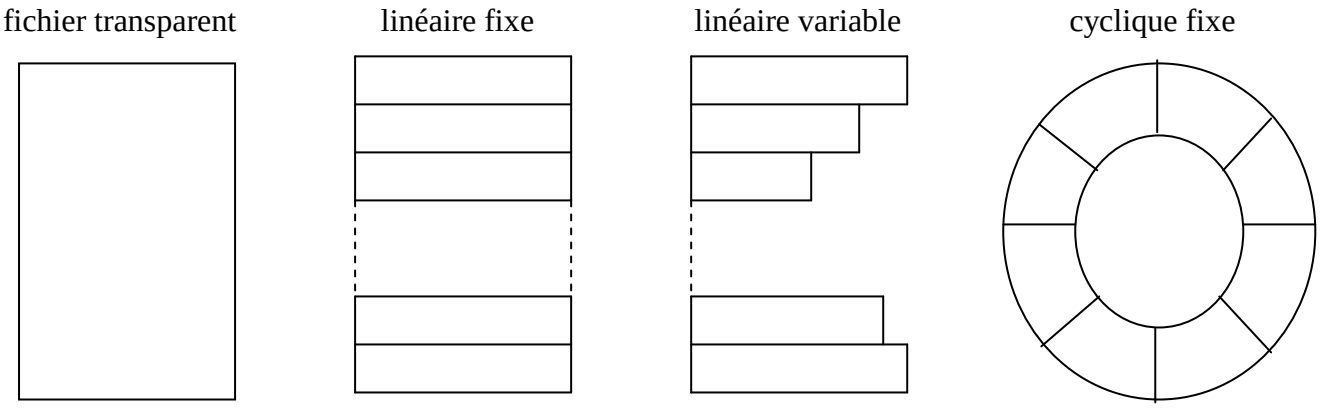
I.7.5.3 dedicated file :

Le DF est un fichier répertoire de la carte à puce qui contient d'autres « dedicated files » et « elementary files ». Un master file est un cas particulier de DF.

I.7.5.4 elementary file :

Un EF est un fichier de données ; il ne peut pas contenir d'autres fichiers. Basé sur la structure de fichier il y a quatre types d'EF. Un fichier transparent est structuré comme une séquence d'octets de données ; alors que les trois autres types d'EF sont structurés comme des séquences d'enregistrements individuellement identifiables. Un fichier linéaire fixe a des enregistrements de taille fixe ; un fichier linéaire variable a des enregistrements de taille variable ; et un fichier cyclique a des enregistrements fixes organisés en anneau.

Dans un fichier cyclique , les enregistrements sont dans l'ordre inverse de l'ordre dans lequel ils ont été insérés dans le fichier – le dernier enregistrement inséré est identifié comme l'enregistrement 1. Quand le fichier devient plein , la prochaine instruction écrit par-dessus le plus ancien enregistrement du fichier et il devient l'enregistrement 1.



I.8 Conclusion :

Dans ce chapitre nous avons donné des notions générales sur les cartes à puce, en abordant l'architecture physique de ces cartes, ainsi que leurs protocoles de communication. Nous avons également donné un aperçu du support logiciel (système d'exploitation) des cartes à puce.

Le chapitre suivant sera consacré à la technologie Java Card.

CHAPITRE 2

Java Card

Chapitre Deux :

Java Card

II.1 JavaCard : introduction et principes :

Vu que les applications étaient longues à développer (cycle de deux ans) et statiques dans leurs fonctions , et vu le besoin en systèmes « programmables » et la recherche de solutions évolutives (dépasser la ROM) ; des propositions ont vu le jour. Parmi celles-ci la Java Card : en novembre 1996, la première proposition d'utilisation de Java pour les cartes est faite par une équipe de Schlumberger (Austin). Cependant le démarrage et produit réel n'a lieu qu'en 1998, et devient une réalité industrielle à partir de 2000. En 2004, le nombre de plates-formes Java Card vendues a atteint le milliard ; en 2008 le nombre a atteint 5 milliards [w3].

II.1.1 étapes de développement de l'industrie :

La carte à microprocesseur et a connu différentes étapes de développement. Les grandes étapes de développement de la technologie sont :

- ✓ Les pionniers (1975-1985): premières idées (les bases technologiques sont établies)
- ✓ 1985-1995: la technologie est améliorée
  - marchés et déploiements importants: CB, GSM
  - limites : besoin de davantage de flexibilité
- ✓ 1995-2005 : explosion du marché, nouveau paradigme avec
  - les cartes évolutives basées sur Java Card
- ✓ 2006: 1,2 billions de téléphones mobiles utilisant des cartes SIM/Java Card
  1,65 billions cartes à puce/ Java Card (source site de Sun)
- ✓ 2005-???: la carte devient un élément du réseau
  - les WebServices

L'arrivée de la technologie Java Card en 1996 était un tournant dans le monde des cartes à puce :
proposition de Java Card API permettant la programmation en Java de la carte : C'est la Java Card 1.0. En 1997, Bull et Gemplus se sont joints à Schlumberger pour créer le Java Card Forum : association regroupant les fabricants de silicium, les encarteurs et des clients et dont les buts sont de :

• Promouvoir la solution de la javacard ,

• Définir et discuter des choix technologiques puis proposer les spécifications à Sun qui en fait le "standard".

En novembre 1997, il y a eu publication de la spécification Java Card 2.0 ; Gemplus démontre en oct./nov. CASCADE, le premier chip RISC 32 bit (ARM 7) avec de la mémoire Flash, une implémentation de la Java Card 2.0 et des DMIs (Direct Method Invocation), etc.

La version 2.0 de la Java Card Specification introduit:

- un environnement d'exécution
- la possibilité d'écrire des applets avec une approche orientée objet (même si le format de chargement n'était pas encore spécifié)

En mars 1999, la version 2.1 qui contient 3 parties est publiée:

- Java Card API Specification
- Java Card Runtime Environment Specification
- Java Card Virtual Machine Specification

II.1.2 place de Java Card dans la plateforme Java :

La technologie Java Card est le plus petit sous-ensemble de la plate-forme Java.

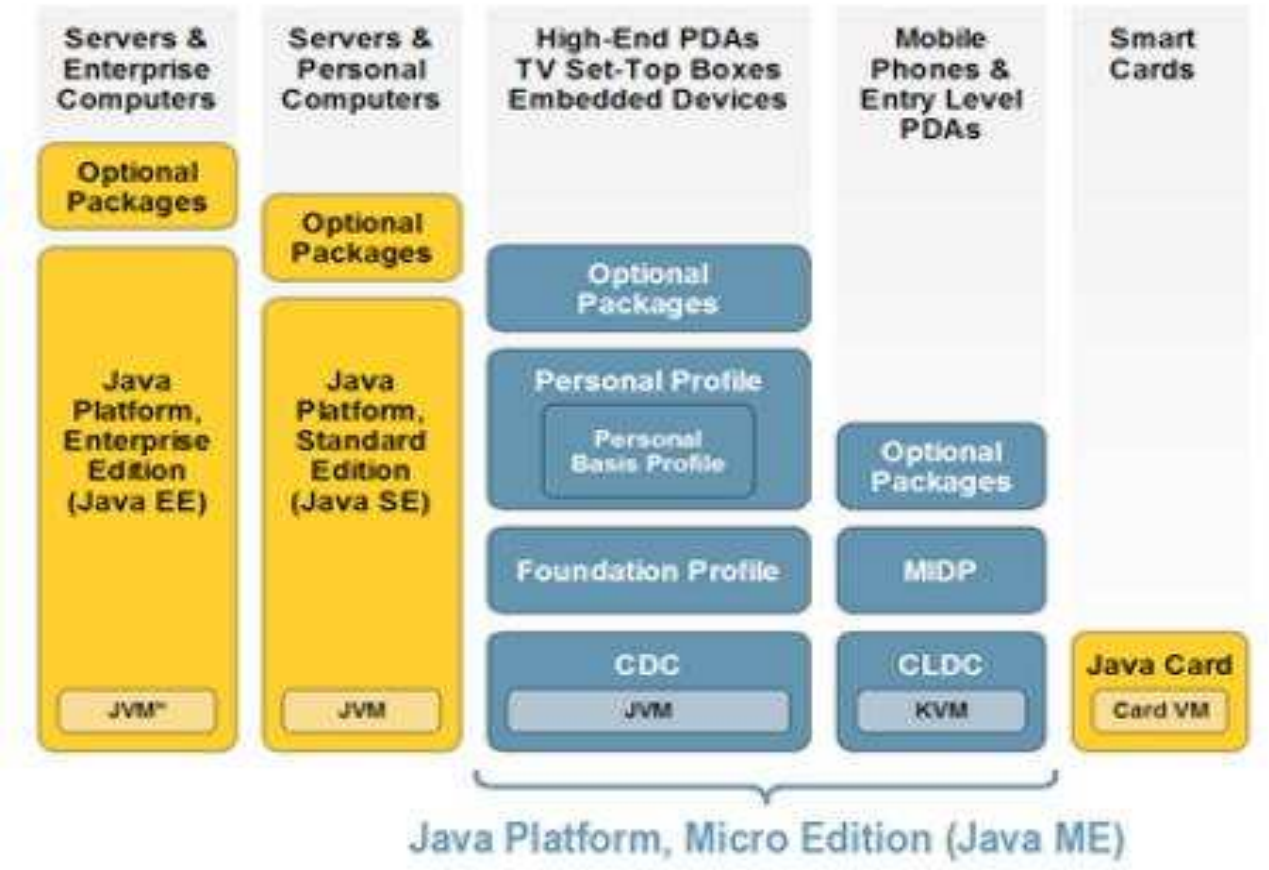


Figure II.a : Java Card par rapport aux plates-formes Java [w3]

Ceci s'explique par le fait de la prise en compte des architectures matérielles des cartes dont les tailles sont très réduites : moins de 1Ko de RAM, 24-28 Ko de ROM et 8 à 16 Ko NVM (EEPROM).

Pour intégrer la technologie Java dans une carte, les choix sont les suivants :

- Réduire les fonctionnalités du langage
- Minima requis pour faire tourner un programme Java Card sont :

24 ko de ROM, 16 ko d'EEPROM et 1 ko de RAM.

D'où la nécessité de distribuer le modèle de la JVM entre le « on Card » et le « off Card ».

Dans les paragraphes suivants, nous décrirons les caractéristiques des plates-formes Java Card 2.2.2 même si depuis Mars 2008 les spécifications Java Card 3.0 ont été publiées par SUN, car jusqu'à présent les implémentations Java Card existantes sont des Java Card 2.1 ou Java Card 2.2. Toutefois, nous consacrerons un paragraphe sur les caractéristiques de Java Card 3.0.

II.1.2.1 caractéristiques non supportées :

• chargement dynamique de classe :

Le chargement dynamique de classe n'est pas supporté dans la plateforme Java Card. Une implémentation de la plateforme Java Car n'est pas capable de charger dynamiquement des classes. Les programmes s'exécutant sur la carte ne peuvent référer que les classes qui existent déjà sur la carte, puisqu'il n'y a aucun moyen de charger les classes durant l'exécution normale du code application.

• gestionnaire de sécurité :

La gestion de la sécurité dans la plateforme Java Card diffère de celle de la plateforme Java.

Dans le standard Java , il y a une classe de gestion de sécurité (`java.lang.SecurityManager`) responsable de l'implémentation des caractéristiques de sécurité. Dans la plateforme Java Card , les politiques de sécurité sont implémentées par la machine virtuelle.

• Ramasse-miettes (garbage collection) :

La technologie Java Card ne nécessite pas de ramasseur de miettes. Non plus que Java Card autorise la désallocation explicite des objets. Aussi les programmeurs d'application ne peuvent pas assumer que les objets qui sont alloués sont désalloués.

• Threads :

La machine virtuelle Java Card ne supporte pas les multiples threads de contrôle. Les programmes Java Card ne peuvent pas utiliser la classe `thread` ou tout autre mot-clé lié aux threads dans le langage de programmation Java.

• Types non supportés :

Les types `long`, `double`, `float`, `char`, `String` ainsi que les opérations sur ces types ne sont pas supportés. Les tableaux multidimensionnels ne sont également pas supportés.

II.1.2.2 caractéristiques supportées :

• Paquetage Java , classes , interface et exceptions :

- Le logiciel écrit pour la plateforme Java Card suit les règles standards pour les paquetages des plateformes Java. Les classes Java Card API sont écrites comme des fichiers source Java, qui incluent des désignations de paquetages. Les mécanismes de paquetage sont utilisés pour identifier et contrôler l'accès aux classes, champs statiques et méthodes statiques.

- Les classes Java peuvent définir ou implémenter des interfaces comme dans le langage de programmation Java. L'invocation des méthodes sur types interface fonctionne comme attendu. Le contrôle de type et l'opérateur `instanceof` fonctionnent aussi correctement avec les interfaces.

- Les programmes Java Card peuvent définir, lancer et capturer des exceptions, comme dans les programmes Java. La classe `Throwable` et ses sous-classes pertinentes sont supportées.

• Héritage , méthode abstraite , surcharge et création d'objets :

Ces caractéristiques sont supportées.

• Types supportés :

Les types boolean , byte et short sont supportés. Les tableaux à une dimension le sont également.

En résumé voici les caractéristiques supportées et non supportées :

Caractéristiques supportées	**Caractéristiques non supportées**
boolean, byte, short	long, double, float, char, String
Tableau à une dimension	Tableau à plusieurs dimensions
Paquetage Java, classes, interface et exceptions	Threads, sérialisation Ramasse-miettes (GC)
Héritage, méthode abstraite, surcharge et création d'objets (instantiation)	Chargement dynamique de classes
« int » est optionnel	Gestionnaire de sécurité

Figure II.b : caractéristiques supportées et non supportées [w3]

On peut remarquer que plusieurs types de données (comme les chaînes de caractères,les nombres flottants) jugés trop gourmands en mémoire ont été supprimés. La notion de multitâche (threads) ne figure pas non plus dans ce langage. En revanche, des méthodes pour gérer des objets persistants et des facilités pour définir des transactions ont été ajoutées.

II.1.3 la plateforme Java Card : [6]

L'architecture de la technologie Java Card est définie par les trois spécifications suivantes :

- la « Java Card Virtual Machine Specification » définit un sous-ensemble du langage de programmation Java et la définition de la machine virtuelle nécessaire pour les applications sur carte à puce ;
- la « Java Card Runtime Environment Specification » décrit précisément le comportement de l'exécution de la Java Card comme la gestion de la mémoire, des applets et d'autres caractéristiques ;
- la « Java Card API Specification » décrit l'ensemble des paquetages et des classes Java nécessaires à la programmation des cartes à puce mais aussi quelques extensions optionnelles.

La figure suivante donne la structure de la Java Card :

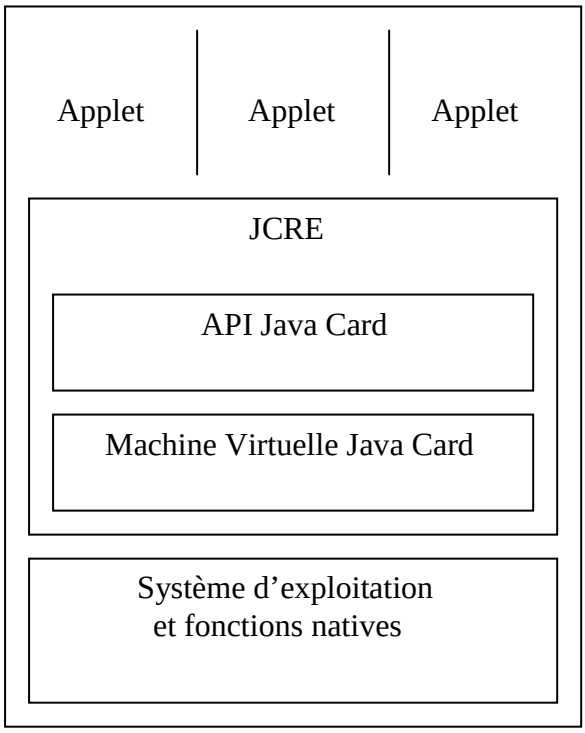


Figure II.c : structure de la Java Card

Les méthodes natives offrent des fonctionnalités de bas niveau telles que les entrées sorties, la gestion de la mémoire, et le co-processeur de cryptographie.

La machine virtuelle Java a pour tâche d'interpréter le bytecode. Elle doit aussi gérer les partages de données entre applications ; car une plateforme Java Card peut héberger plusieurs applications : il faut donc gérer la confidentialité des données.

Le framework (API) Java Card est conçu pour faciliter la programmation d'applications. Il masque les détails de la structure de la carte, et fournit aux développeurs une interface de programmation facile à manipuler.

La spécification du Java Card Runtime Environment intègre les deux spécifications précédentes plus des modules complémentaires.

Tout applet Java Card est une instance d'une classe qui étend la classe Applet. Cette classe contient quelques méthodes qui doivent être surchargées durant la conception de l'applet.

D'un point de vue global, une plateforme Java Card peut être représentée comme un système comportant deux zones différentes. La première est figée. Elle est constituée du système d'exploitation et des extensions éventuellement fournies par les fournisseurs de la carte. Elle contient le JCRE et le système d'exploitation sous-jacent. La deuxième est modifiable. Elle est constituée des applets chargés dans la carte. Les interfaces utilisateurs de ces deux entités sont donc universelles et identiques pour toutes les cartes implantant la technologie Java Card.

Les applets s'exécutent au dessus de ce processeur virtuel et demandent des ressources et des services systèmes au JCRE à travers les APIs qui peuvent être vues comme des bibliothèques universelles présentes sur toutes les Java Cards.

II.1.4 la machine virtuelle (JCVM) :

La machine virtuelle Java Card (JCVM) a une architecture comparable à celle de la machine virtuelle Java (JVM) ; la différence est que la JCVM est découpée en deux parties : [6], [5]

- une partie embarquée sur la carte qui inclut l'interpréteur de byte code ;
- une partie hors carte qui tourne sur une station de travail et qui comprend le convertisseur.

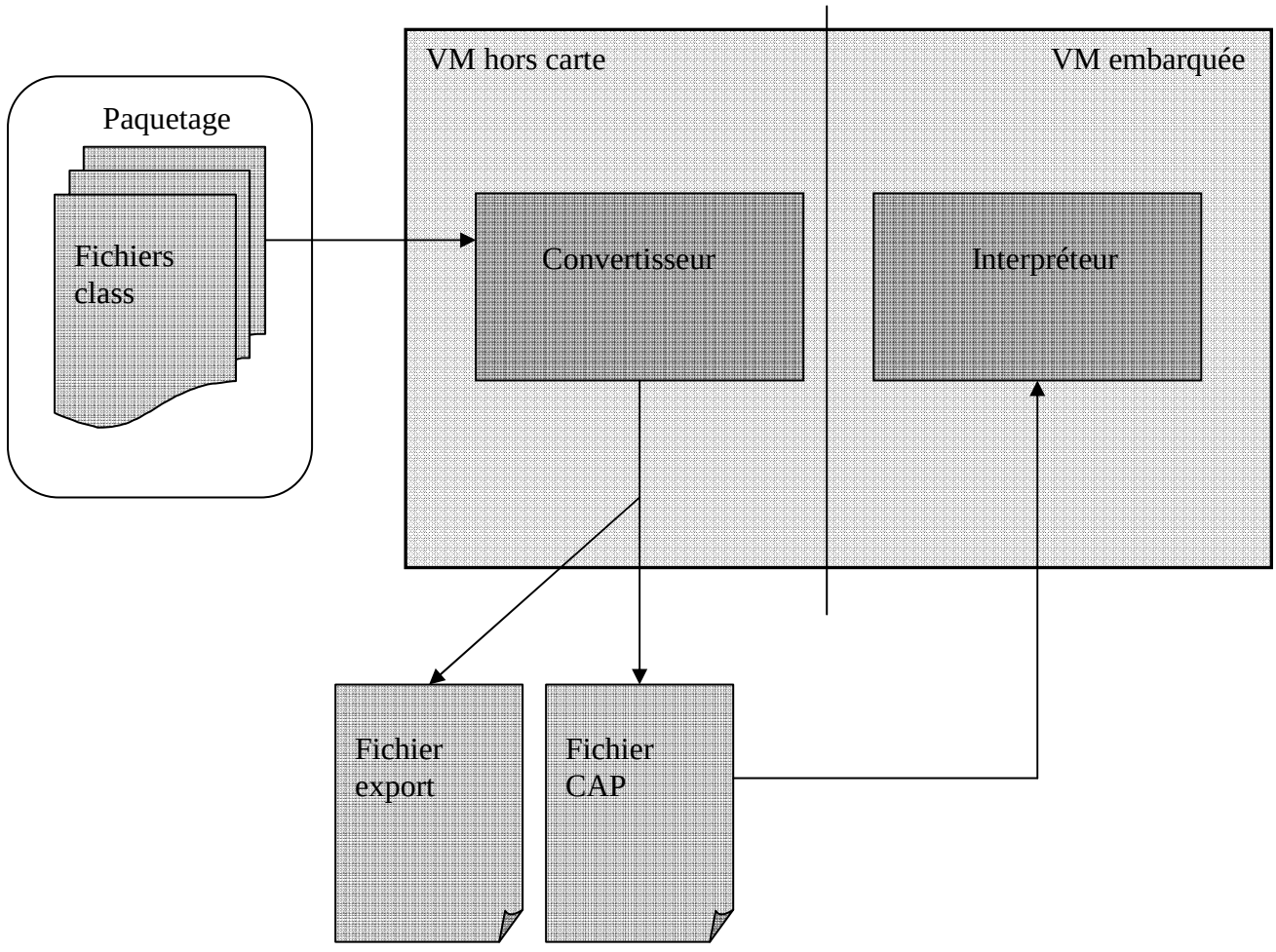


Figure II.d : la machine virtuelle Java Card

Ensemble, ces deux parties implantent toutes les fonctions d'une machine virtuelle.
Le convertisseur charge et traite les fichiers class qui composent le paquetage Java à convertir et produit en sortie un fichier CAP (Converted APplet) qui est une représentation binaire exécutable des classes. En plus de la création du fichier CAP, le convertisseur génère également un fichier export représentant les APIs publiques du paquetage converti.
Par ailleurs, toute caractéristique non supportée du langage utilisée dans une applet est détectée par le convertisseur et conduit à un échec du processus de conversion. Si la conversion réussit, le fichier CAP peut alors être chargé sur la carte à puce afin d'y être exécuté par l'interpréteur.

A cause du découpage de la JCVM, la plateforme est distribuée entre la carte à puce et la machine de développement, et ce, dans le temps et dans l'espace.

II.1.4.1 Fichier CAP et fichier export :

La technologie Java Card introduit deux nouveaux formats de fichiers binaires qui apportent l'indépendance de la plateforme de développement, de la distribution et de l'exécution de logiciels Java Card. Ce sont : [5]

- le fichier CAP
- le fichier export

Un fichier CAP contient une représentation binaire exécutable des classes d'un paquetage Java. Ce fichier CAP est une archive utilisant le format de fichiers JAR pour stocker un ensemble de composants. Chacun de ces composants est stocké comme un fichier individuel dont le nom a pour extension .cap. Le format du fichier de chacun de ces composants suit le modèle suivant :

Figure II.e : format de fichier d'un composant du paquetage

Chaque composant décrit un aspect du contenu d'un fichier CAP. Aujourd'hui avec les spécifications, il y a douze composants officiels dont trois sont optionnels. Mais, il existe aussi la possibilité de créer de nouveaux composants pour des besoins spécifiques des vendeurs.
Voici la liste des composants officiels avec leurs étiquettes et une brève description :
- header (1) : contient le « nombre magique » et les numéros de version de l'implantation de la machine virtuelle Java Card pouvant supporter le paquetage ;

- directory (2) : contient la liste des autres composants avec leurs taille respective ;

- applet (3) (optionnel) : contient la liste des applets avec leurs AID et une référence (i.e un offset) sur la méthode install() (i.e la première méthode appelée par l'interpréteur pour installer l'applet) ;

- import (4) : contient la liste des paquetages référencés dans ce paquetage

- constant pool (5) : contient la table de références aux classes, aux méthodes et aux champs, respectivement dans les composants de classes, de méthodes et de champs ;

- class (6) : contient la table des classes et interfaces ;

- méthode (7) : contient le code des méthodes des classes

- static field (8) : contient les valeurs initiales de tous les champs statiques du paquetage ;

- reference location (9) : contient la liste des références relatives dans le code des méthodes à transformer en références absolues au chargement, en vue de procéder à des vérifications dans le code ;

- export (10) (optionnel) : contient la liste des éléments de ce paquetage que les autres paquetages peuvent référencer directement (méthodes et champs statiques public ou protected dans les classes publiques) ;

- descriptor (11) : contient les informations de typage (i.e signatures des méthodes) si ce paquetage est accessible par d'autres paquetages ;

- debug (12) (optionnel) : contient toutes les méta-données nécessaires pour déboguer un paquetage dans une JCVM adaptée. Il n'est pas requis pour exécuter des programmes hors d'un environnement de débogage.

Dans la technologie Java, le fichier class est la pièce centrale de l'architecture Java. Il définit le standard pour la compatibilité binaire de la plate-forme Java.
Dans la technologie Java Card, à cause des caractéristiques distribuées de l'architecture système, c'est la fichier CAP qui est le format standard de fichier de compatibilité binaire. Si un fichier CAP ne contient pas d'applet (i.e aucun fichier class n'hérite de javacard.framework.Applet) alors le composant applet est absent et il s'agit d'une bibliothèque.

Les fichiers export ne sont pas chargés dans la carte et ne sont pas directement utilisés par l'interpréteur. Ils sont produits et utilisés par le convertisseur dans des buts de vérification et d'édition de liens. Ainsi, les fichiers peuvent être vus comme les fichiers entêtes dans la programmation en langage C. Il sert au développeur, client de bibliothèque, pour la phase d'édition entre les bibliothèques utilisées et les applets/bibliothèques qu'il développe. Un fichier export contient donc les informations publiques sur les APIs pour un paquetage de classes complet. Il définit l'étendue et le nom des classes et l'étendue et la signature des méthodes et des champs de classes. En revanche un fichier export ne contient pas d'information relative à l'implantation (i.e pas de bytecode). Ce fichier peut donc être librement distribué par un développeur d'applet aux utilisateurs potentiels de l'applet ou de la bibliothèque sans révéler les détails de l'implantation interne.

II.1.4.2 Convertisseur Java Card :

Contrairement à la machine virtuelle Java, qui travaille sur une classe à la fois, l'unité de conversion en Java Card est le paquetage [6]. Les fichiers class sont produits par le compilateur Java depuis le code source. Le convertisseur examine ensuite les fichiers class qui composent le paquetage Java et convertit ce paquetage en un fichier .CAP.

Durant la conversion, le convertisseur réalise des tâches qu'une machine virtuelle Java classique (i.e sur une station de travail) doit réaliser au chargement :

- vérifier que les images de classes sont bien formées ;
- contrôler les violations du langage Java Card ;
- réaliser les initialisations des variables statiques ;
- résoudre les références symboliques aux classes, méthodes et champs dans la forme la plus compacte qui peut être traitée efficacement sur la carte ;
- optimiser le bytecode en tirant avantage des informations obtenues au chargement des classes et à l'édition des liens ;
- allouer l'espace et créer les structures de données de la machine virtuelle pour représenter les classes.

Grâce aux tâches réalisées par le convertisseur, la partie de la machine virtuelle embarquée sur la carte peut être plus petite et plus performante, car déchargée d'une partie de son travail.

En fait le convertisseur ne prend pas seulement des fichiers class à convertir mais aussi un ou plusieurs fichiers export. En plus de la production du fichier CAP, le convertisseur génère également un fichier export pour le paquetage converti. La figure II.f montre comment un paquetage est converti. Le convertisseur charge toutes les classes d'un paquetage Java. Si le paquetage importe des classes provenant d'autres paquetages, le convertisseur charge aussi les fichiers export de ce paquetage. En effet, comme nous l'avons vu les fichiers export contiennent des informations sur les paquetages chargés sur la carte qui

vont servir au convertisseur pour réaliser son travail.

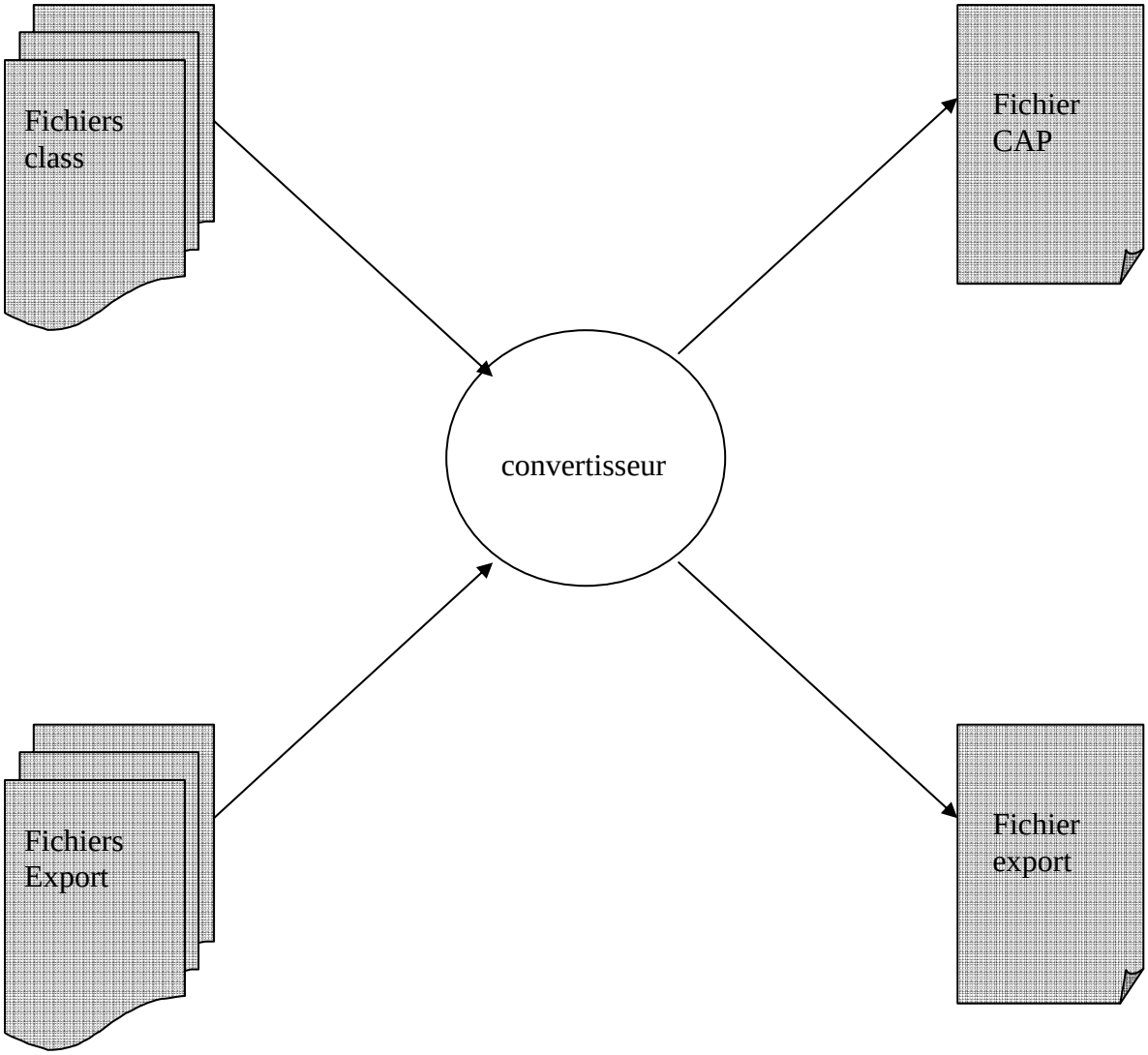


Figure II.f : conversion d'un paquetage [6]

II.1.4.3 Interpréteur Java Card :

L'interpréteur Java Card fournit le support d'exécution du modèle du langage Java qui autorise une indépendance du code de l'applet par rapport au matériel. L'interpréteur est donc en quelque sorte le processeur virtuel « universel » pour la Java Card. Il réalise les tâches suivantes : [6]

- il exécute les instructions du bytecode et donc les applets ;
- il contrôle les allocations de mémoire et les créations d'objets ;
- il joue un rôle crucial pour assurer la sécurité lors de l'exécution.

II.1.4.4 Installeur Java Card et programme d'installation hors carte :

L'interpréteur Java Card ne charge pas le fichier CAP lui-même. Il exécute seulement le code du fichier CAP. Dans la technologie Java Card, le mécanisme de téléchargement et d'installation du fichier CAP est contenu dans une unité appelée l'installeur.

L'installeur Java Card réside sur la carte et il coopère avec un programme d'installation situé hors de la carte. Ce programme transmet à l'installeur qui s'exécute sur la carte, via le terminal (i.e CAD), l'exécutable binaire contenu dans un fichier CAP. L'installeur écrit le fichier binaire dans la mémoire de la carte à puce, le lie avec les autres classes déjà présentes sur la carte, puis crée et initialise les structures de données qui sont utilisées par le Java Card Runtime Environment (JCRE) [6] [5]. Ce processus est illustré en figure II.g.

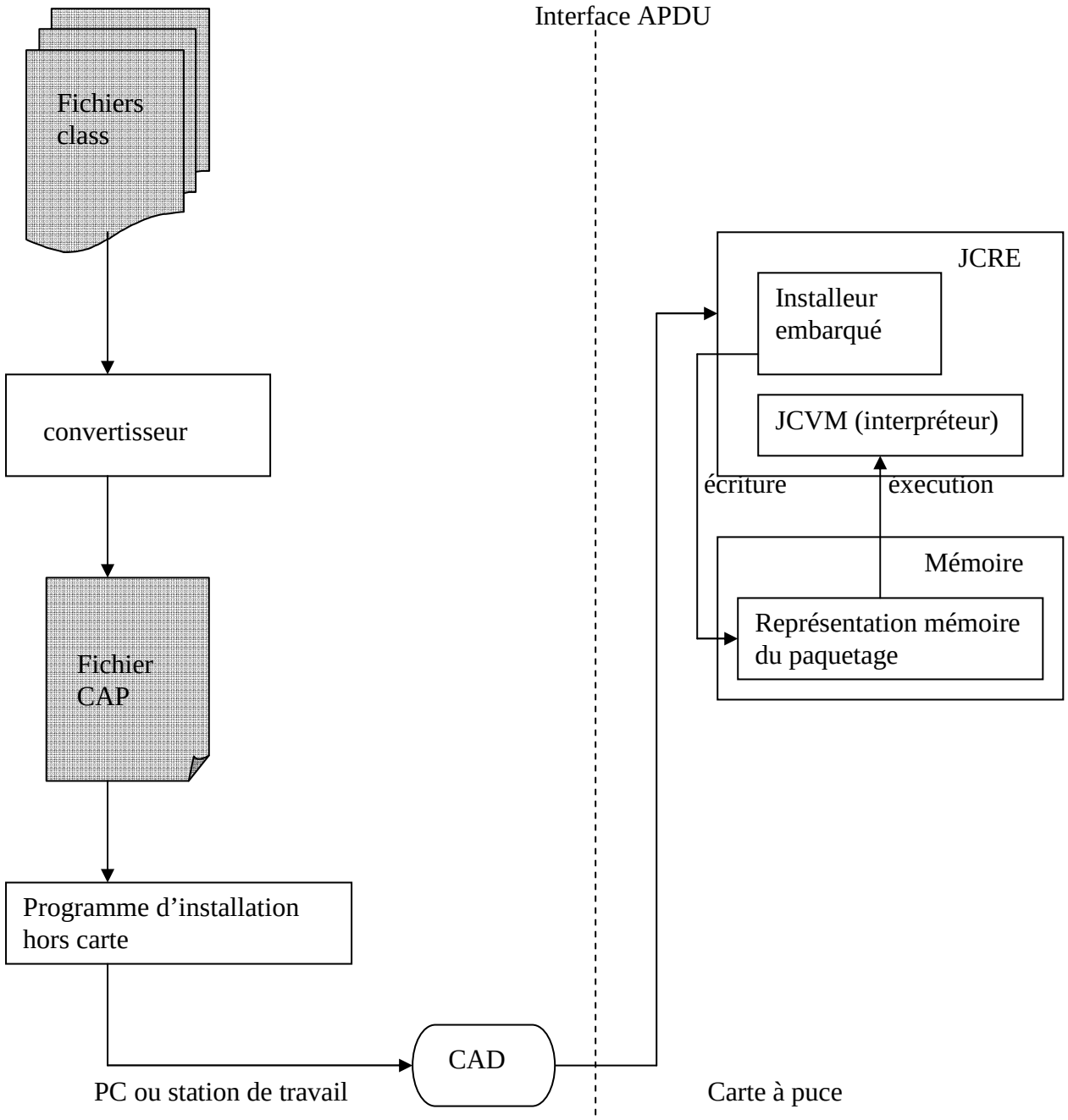


Figure II.g : Installeur Java Card et programme d'installation hors carte

La séparation entre interpréteur et l'installeur de fichier CAP permet à l'interpréteur de rester compact et offre une grande flexibilité dans l'implantation de l'installeur. Par exemple, ce dernier peut être écrit comme une applet Java Card ou non.

Les spécifications Java Card précisent que l'installeur est un composant optionnel mais s'il est absent une Java Card ne pourra pas accepter le chargement des applications après son émission : toutes les applets doivent alors être écrites dans la mémoire de la carte à puce lors de sa fabrication.

Par ailleurs, la description de l'installeur dans les spécifications est assez brève et ce n'est que depuis les versions 2.2 des spécifications que des remarques sur la possibilité d'effacer des applets ont fait leur apparition. En effet, les spécifications ne détaillent que peu la gestion des applets sur la carte (i.e « comment les charger ?», « comment les effacer ? », « quel est leur cycle de vie », etc.). Toutes les parties concernant l'installeur (i.e le chargement, l'installation et l'effacement) sont donc plus des recommandations minimales qu'il faudrait suivre, plutôt que des spécifications visant à standardiser ce composant.

Ainsi, on ne trouve, par exemple, aucune standardisation des APDUs pour le chargement, l'installation ou l'effacement dans les spécifications du JCRE.

II.1.5 l'environnement d'exécution (JCRE) :

L'environnement d'exécution de la Java Card (JCRE) représente l'ensemble des composants du système Java Card présents à l'intérieur de la carte à puce. Le JCRE est responsable de la gestion des ressources de la carte, de la communication réseau, de l'exécution des applets, du système de la carte et de la sécurité des applets [6].

Le JCRE est une sorte de système d'exploitation « universel » pour la Java Card. Il apporte l'indépendance des applets par rapport aux technologies propriétaires des vendeurs de carte à puce en leur fournissant un système et des APIs standards. Il en résulte que les applets sont plus faciles à écrire et sont portables sur diverses architectures de carte à puce.

Comme on peut le voir en figure II.h, le JCRE se situe au dessus du matériel de la carte à puce et du système natif. Il englobe :

- les méthodes natives ;
- la JCVM (l'interpréteur de byte code) ;
- les classes système du JCRE ;
- les APIs Java Card ;
- des extensions spécifiques à l'industrie ;
- l'installeur.

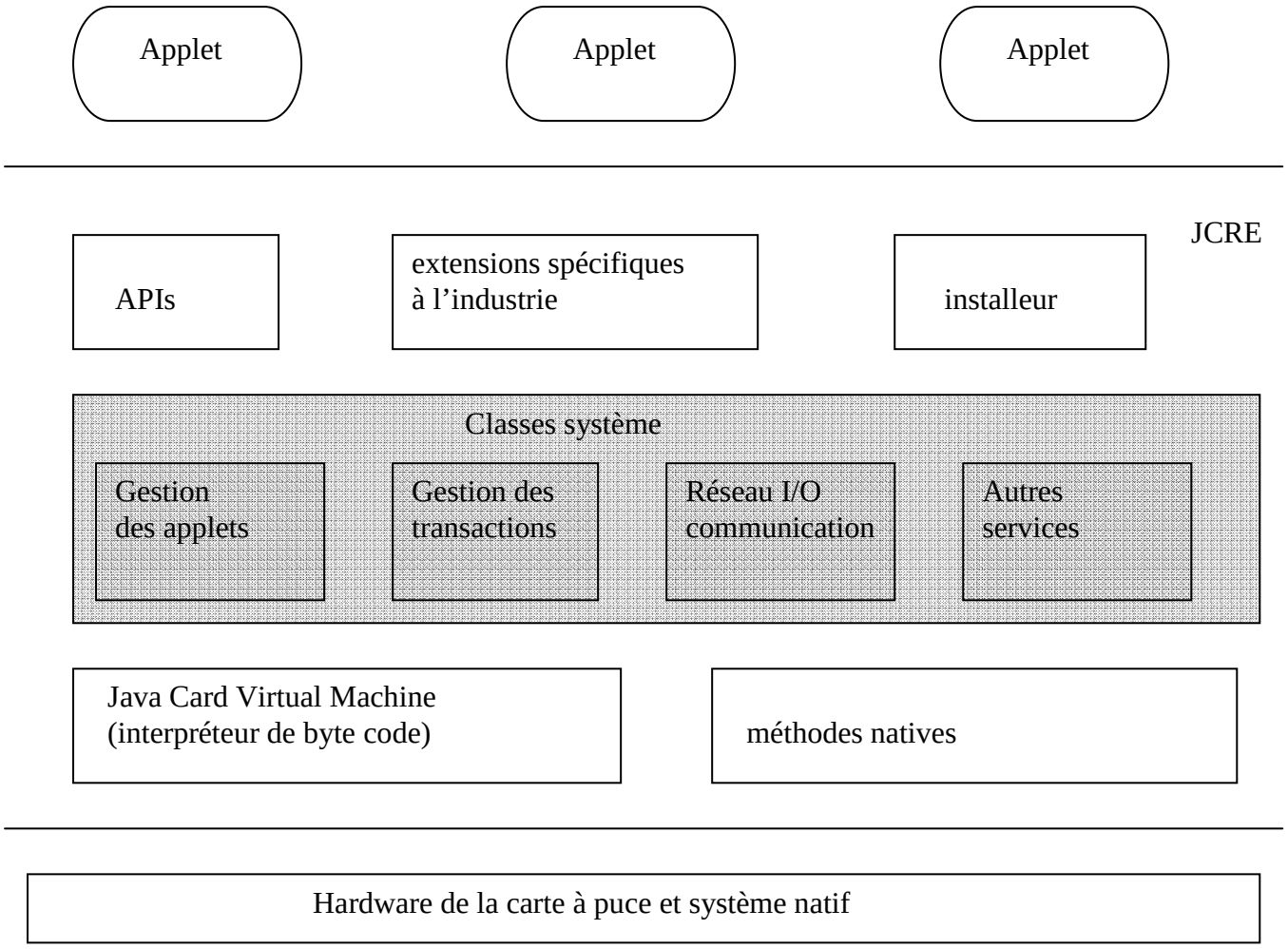


Figure II.h : architecture du système dans la carte

La couche basse du JCRE comprend les méthodes natives et la machine virtuelle Java Card. Les méthodes natives procurent un accès aux services bas niveau de la puce pour la JCVM et pour la couche supérieure que forment les classes système du JCRE. Ces méthodes natives sont responsables du traitement des protocoles de communication de bas niveau, de la gestion mémoire, du support de la cryptographie, …etc [6].

Les classes système sont la partie essentielle du JCRE. Elles sont analogues au noyau d'un système d'exploitation et elles ont en charge :

- la gestion des transactions ;
- la gestion des communications entre les applications hôtes (applications qui tournent sur la machine reliée au terminal) et les applets Java Card ;
- le contrôle de la création, sélection et désélection des applications.

Pour réaliser ces tâches, les classes système se basent généralement sur le système natif de la carte via les méthodes natives présentées ci-dessus.

Les classes APIs Java Card sont compactes et optimisées pour le développement d'applets pour les cartes à puce. Les développeurs peuvent donc concentrer leurs efforts sur les détails de leurs applets plutôt que sur les détails 1'infrastrucure du système des cartes à puce. Les applets accèdent aux services du JCRE à travers les APIs Java Card.

Les extensions spécifiques à l'industrie sont des bibliothèques chargées de fournir des services supplémentaires ou de redéfinir la sécurité et le modèle système. Par exemple, ajouter des services au JCRE pour la gestion des applications issues de partenaires différents. Comme nous l'avons déjà vu, l'installeur est un composant optionnel du JCRE et c'est lui qui permet le chargement des applets sur la carte après leur émission.

Les applets Java Card sont des applications sur la plateforme Java Card. Elles sont évidemment écrites dans le sous-ensemble du langage Java décrit et sont contrôlées et gérées par le JCRE. En effet, elles sont téléchargeables via le processus d'installation mais elles interagissent aussi de façon très étroite avec les différents services du JCRE.

II.1.5.1 les caractéristiques du JCRE :

En plus du support du modèle d'exécution du langage Java, le JCRE supporte quatre autres caractéristiques d'exécutions :

- Les objets persistants et temporaires (*transient* en anglais) : [6]

Par défaut les objets Java Card sont persistants et sont donc créés dans la mémoire persistante. L'espace et les données de tels objets existent à travers les sessions CAD.

Pour des raisons de sécurité et de performance, les applets peuvent également créer des objets dont les données seront placées en mémoire volatile. De tels objets sont appelés temporaires (*transient* en anglais). Ces objets contiennent des données temporaires qui ne persisteront pas à travers les sessions CAD.

- Opérations et transactions atomiques : [6]

La JCVM assure que chaque opération d'écriture dans un champ d'objet ou d'une classe est atomique. Donc, en cas de problème lors de l'écriture, le champ mis à jour prend soit la nouvelle valeur soit il est restauré avec la valeur précédente. Le JCRE propose également des mécanismes de transactions via les APIs. Ainsi une applet peut réaliser plusieurs écritures durant une transaction ce qui permet de s'assurer que tout est mis à jour dans une transaction complète, soit rien n'est réalisé si une erreur apparaît au milieu de la transaction.

- Le pare-feu entre les applets et mécanismes de partage : [6]

Le pare-feu isole chaque applet des autres applets à l'intérieur de son propre espace appelé « contexte ». Ainsi l'existence et les opérations d'une applet n'ont pas d'effet sur les autres applets présentes sur la carte.

Ce pare-feu est renforcé par la JCVM puisque c'est elle qui exécute les bytecodes. Par ailleurs dans les situations où les applets ont besoin de partager des données ou d'accéder à des services du JCRE, la machine virtuelle permet l'utilisation de telles fonctions à travers des mécanismes sécurisés de partage.

- Le service d'invocation de méthodes à distance : [5]

Depuis la version 2.2 de Java Card, les spécifications ont introduit un sous-ensemble de RMI appelé JCRMI (Java Card Remote Method Invocation). Il fournit à une application cliente fonctionnant côté CAD, un mécanisme pour appeler une méthode sur un objet distant présent sur la carte. Ce service a pour objectif de faciliter le développement d'applications clientes en utilisant le protocole RMI plutôt que les APDUs. Cela permet de cacher la complexité sous-jacente du passage d'argument et de la pile protocolaire entre l'application cliente hors-carte et l'application serveur sur la carte. Un paquetage embarqué sur la Java Card assure la gestion de la couche transport pour JCRMI.

II.1.5.2 les APIs :

Les APIs Java Card sont un ensemble de classes optimisées pour la programmation des cartes à puce en accord avec le modèle ISO 7816 [6]. Beaucoup de classes de la plateforme Java (ex. celles relatives au réseau, à l'affichage, ..etc) ne sont pas nécessaires et donc pas disponibles sur la plateforme Java Card. Jusqu'à la version 2.1.1 des spécifications Java Card les APIs ne comprenaient qu'un noyau de trois paquetages (`java.lang`, `javacard.framework` et `javacard.security`) et un paquetage d'extension (`javacardx.crypto`). Mais depuis la version 2.2 trois autres paquetages ont été ajoutés pour supporter JCRMI : `java.io`, `java.rmi` et `javacard.framework.service`.

a) le paquetage `java.lang` :

Le paquetage `java.lang` est, sur la platefome Java Card, un sous-ensemble strict de son équivalent sur la plateforme Java. Ce paquetage supporte les classes `Object`, `Throwable` et `Exception`. C'est lui qui fournit les fondements pour le support du langage Java. La classe `Object` est définie comme la classe racine de la hiérarchie des classes sur la plateforme Java Card. Les classes d'exceptions fournies permettent également d'assurer une sémantique cohérente lorsqu'une erreur apparaît à cause d'une violation du langage de programmation Java. Par exemple, la machine virtuelle Java et celle Java Card lanceront toutes les deux `NullPointerException` quand on voudra accéder à une référence `null`.

b) le paquetage `javacard.framework` :

La classe la plus importante qu'il définit est la classe de base `Applet`.

Ce paquetage apporte les classes et les interfaces essentielles pour programmer des applets Java Card qui fournit la structure sous-tendant le modèle d'exécution de l'applet en interaction avec le JCRE pour toute

la durée de vie de l'applet. Il fournit également la classe APDU nécessaire pour gérer les communications des applets avec l'extérieur. Depuis Java Card 2.2, c'est aussi dans ce paquetage que se trouve l'interface MultiSelectable qui indique si l'applet accepte les canaux logiques et la multi sélection. Puisque la classe `system` du paquetage `java.lang` de la plateforme Java n'est naturellement pas supportée, la plateforme Java Card propose dans le paquetage `javacard.framework` la classe `JCSystem` afin d'accéder aux services du JCRE.

Cette classe permet notamment de gérer les mécanismes de transaction et de partage d'objets au travers du pare-feu en association avec l'interface `Shareable` mais aussi de créer des objets temporaires et d'appeler le mécanisme de ramasse-miettes si celui-ci est présent.

Enfin, ce paquetage fournit d'autres classes comme la classe `OwnerPIN` qui propose une implantation de référence du mécanisme de PIN. Les deux autres classes restantes sont `AID` pour faire des opérations sur les AIDs et `Util` qui propose des méthodes souvent utilisées dans la programmation en Java Card.

c) le paquetage `javacard.security` :

Le paquetage `javacard.security` fournit une architecture aux fonctions cryptographiques supportées sur la plateforme Java Card. Sa structure est basée le paquetage **`javacard.security`** de la plateforme Java. Il propose une classe `KeyBuilder` pour fabriquer des clés cryptographiques de différents types et des interfaces pour les manipuler. Il met principalement à disposition du programmeur les classes abstraites `RandomData`, `Signature`, `MessageDigest`, et `Checksum` pour générer respectivement des nombres aléatoires, des signatures, des empreintes et des sommes de contrôle. Il définit également l'exception `CryptoException` pour la gestion des différentes erreurs.

d) le paquetage `javacardx.crypto` :

Le paquetage `javacardx.crypto` est un paquetage d'extension. Il contient les classes cryptographiques et les interfaces qui sont sujettes à une demande pour l'exportation aux Etats-Unis. Il définit une classe abstraite de base `Cipher` qui supporte les fonctions de chiffrement et de déchiffrement pour les différents algorithmes implantés sur la carte. En effet, dans le domaine de la cryptographie, Java Card n'impose pas aux vendeurs d'implanter tous les algorithmes cryptographiques, ni même un ensemble minimal. On peut ainsi se retrouver avec des Java Cards aux possibilités bien différentes dans les domaines cryptographiques.

e) le paquetage `java.io` :

Ce paquetage est un sous-ensemble strict de celui de la plateforme Java. Il ne définit qu'une exception `IOException` pour indiquer un problème de communication. Celle-ci n'est présente que pour les besoins de JCRMI qui en hérite au travers de la classe `RemoteException` du paquetage `java.rmi`.

f) le paquetage `java.rmi` :

Ce paquetage définit l'interface `Remote` qui identifie les interfaces dont les méthodes peuvent être appelées par des applications clientes du CAD. Il définit également l'exception `RemoteException` qui pourra être lancée pour indiquer qu'une exception s'est produite lors de l'exécution d'un appel de méthode distante.

g) le paquetage `javacard.framework.service` :

Ce paquetage fournit un ensemble de classes et d'interfaces qui permettent à une applet Java Card d'être conçue comme le regroupement de plusieurs services. Il fournit aussi une classe d'agrégation appelée `Dispatcher` qui propose des méthodes pour ajouter et retirer des services de sa base interne, et pour router les commandes APDUs vers les services enregistrés.

Le paquetage contient également l'interface `Service` que doivent implanter tous les services. Cette interface propose toutes les méthodes nécessaires pour traiter les commandes APDUs. Il existe aussi deux sous-interfaces de `Services`, `RemoteService` et `SecurityService` pour l'implantation de services aux fonctionnalités plus spécifiques. `RemoteService` est utilisé pour définir des services qui permettent à des processus distants d'accéder aux services présents sur la Java Card alors que `SecurityService` est utilisé pour définir des services qui fournissent des méthodes pour connaître le niveau de sécurité actuelle (ex. intégrité des données entrantes, confidentialité …etc.).

La classe `BasicService` fournit une implantation par défaut des méthodes définies dans l'interface `Service` et par conséquent, la fonctionnalité de base d'un service. Tous les services seront donc des sous-classes de `BasicService`.

Par ailleurs, afin de permettre à un programme Java fonctionnant côté poste client d'appeler des méthodes sur les objets distants d'une applet Java Card, le paquetage fournit les classes `CardRemoteObject` et `RMIService`. Ces classes contiennent donc les fonctionnalités minimales exigées pour permettre l'invocation de méthodes à distance pour la plateforme Java Card (JCRMI).

II.2 Développement d'une applet :

Nous parlerons ici du développement des applets après la phase de fabrication de la carte. A la différence des applets gravées dans la ROM et fournies par le producteur de la carte, elles ne peuvent pas accéder à des méthodes natives, écrites en langages de plus bas niveau comme C ou assembleur.

II.2.1 Applets Java Card :

Une applet Java Card étend toujours `javacard.framework.Applet`, qui est la superclasse pour toutes les applets. L'applet sur la carte est une instance de classe applet, un objet persistant, qui une fois créé réside sur carte en permanence.

Une applet qui consiste en plus d'une classe peut être organisée, selon sa complexité, en un ou plusieurs paquetages.

Chaque applet possède un identifiant unique AID spécifié par la norme ISO 765-5. Un AID est représenté par un tableau dont la taille varie entre 5 et 16 octets. Il est constitué par la concaténation du RID (Ressource Identifier de taille fixe de 5 octets) et du PIX (Proprietary Identifier eXtension de taille variant entre zéro et onze octets). C'est l'ISO qui procède à l'attribution des RIDs (5 octets) aux entreprises afin que chacune ait son propre RID et ensuite chaque entreprise contrôle la gestion des PIXs.

Une applet est inactive après son installation. Elle doit être explicitement sélectionnée pour devenir active.

Une applet active attend les commandes APDU puis répond et ce jusqu'à ce qu'une autre applet soit sélectionnée (ou la carte retirée du lecteur).

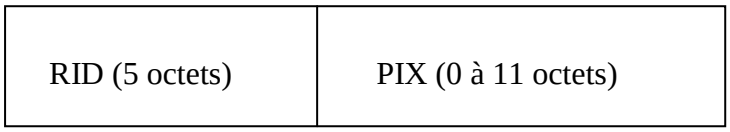


Figure II.i : identifiant d'application AID

Chaque applet et chaque paquetage possède un AID unique. Le RID identifiant le fournisseur de l'applet, les applets définis dans un paquetage partagent le même RID que celui-ci. L'AID d'un paquetage et celui par défaut de chaque applet de ce paquetage sont spécifiés dans le fichier CAP. On parle d'AID par défaut pour une applet car il est possible de modifier , à l'installation , l'AID qui était le sien dans le fichier CAP à condition de conserver le même RID. Les AIDs sont fournis au convertisseur lorsque le fichier CAP est généré.

II.2.2 La structure minimale d'une applet :

Une applet doit au minimum implémenter les méthodes suivantes :

- install() : installe l'applet et initialise les objets ;
- select() : à implémenter s'il est nécessaire d'initialiser ou de personnaliser les sessions ;
- deselect() : peut être implémentée pour personnaliser la fin de session ;
- process() : gère les commandes APDU entrants.

II.2.3 Le cycle de vie d'une applet :

Le cycle de vie d'une applet est étroitement lié au fonctionnement de la JCVM.

- Compilation des classes d'un programme Java Card en code binaire ;
- Vérifications statistiques, simplifications, conversion au format CAP ;
- Simulation et débogage sur un émulateur JCRE ;
- Téléchargement d'une applet sur la carte sous forme d'un fichier CAP ;

- Installation : l'instanciation de l'applet ;
- Eventuellement, fonctionnement limité pour besoins de configuration de la carte ;
- Personnalisation : initialisation de certains paramètres d'instance ;
- Fonctionnement : appel d'une instance ;
- Suppression de l'applet.

II.2.4 Les étapes de développement d'une applet :

Quatre étapes essentielles constituent la phase de développement d'une applet :

- Spécifier les fonctions de l'applet
- Assigner des AIDs à l'applet et au paquetage contenant la classe de l'applet
- Concevoir les programmes de l'applet
- Définir l'interface entre l'applet et le terminal

II.2.4.1 Spécification les fonctions de l'applet :

Cette étape consiste à spécifier toutes les fonctions rendues par l'applet et satisfaisant. A titre d'exemple pour réaliser une applet porte-monnaie qui va stocker de la monnaie électronique on spécifiera que l'applet supportera les fonctions de crédit, débit, contrôle de la balance ou consultation, et un code PIN pour parer à toute éventuelle utilisation frauduleuse de la carte.

II.2.4.2 Spécification des AIDs :

Dans la technologie Java Card, chaque applet est identifiée et sélectionnée par un identificateur (AID). De même à chaque paquetage Java est assigné un AID. Cette convention de nom est conforme à la spécification de la carte à puce définie dans l'ISO 7816.

II.2.4.3 Conception des programmes de l'applet :

Consiste à définir les méthodes, les classes et les interfaces constituant l'applet.

Une applet Java Card doit étendre la classe `javacard.framework.Applet`. Cette classe est une superclasse des applets résidents sur la carte. Elle définit les méthodes courantes que doit utiliser une applet pour interagir avec le JCRE, l'environnement d'exécution.

La classe `javacard.framework.Applet` fournit un framework pour l'exécution des applets. Les méthodes définies dans la classe sont appelées par le JCRE lorsque celui-ci reçoit des commandes APDU à partir du lecteur (CAD : Card Acceptance Device).

Une fois le code de l'applet proprement chargé sur la carte et lié aux autres paquetages se trouvant sur la carte, la vie de l'applet commence lorsqu'une instance de l'applet est créée et enregistrée au sein du JCRE.

Résumé de la méthode	
`public void`	`deselect()` Appelée par le JCRE pour informer l'applet courante qu'une autre (ou la même) applet sera sélectionnée
`public Shareable`	`getShareableInterfaceObject (AID client AID, byte parameter)` Appelée par le JCRE pour obtenir un objet interface partageable de cette applet serveur au nom d'une requête d'une applet cliente
`public static void`	`install (byte[] bArray, short bOffset, byte bLength)` Le JCRE appelle cette méthode statique pour créer une instance de la sousclasse d'`Applet`
`public abstract void`	`process (APDU apdu)` Appelée par le JCRE pour traiter une commande APDU qui arrive
`protected final void`	`register ()` Cette méthode est utilisée pour enregistrer cette instance d'applet au sein du JCRE et assigner l'AID par défaut dans le fichier CAD à l'instance de l'applet
`protected final void`	`register (byte[] bArray, short bOffset, byte bLength)` Cette méthode est utilisée par l'applet pour enregistrer cette instance d'applet au sein du JCRE et assigner un AID spécifié dans le tableau `bArray` à l'instance de l'applet
`Public boolean`	`select ()` Appelée par le JCRE pour informer cette applet qu'elle a été sélectionnée
`protected final Boolean`	`selectingApplet ()` Cette méthode est utilisée par la méthode `process ()` de l'applet pour distinguer la commande `SELECT APDU` qui a sélectionné cette applet de toutes les autres commandes APDU `SELECT APDU` qui peuvent se rapporter à un fichier ou à la sélection d'un état interne de l'applet

Figure II.j : Méthodes public et protected définies dans la classe `javacard.framework.Applet`[w3]

Une applet doit définir la méthode `install()` pour créer une instance d'applet et doit enregistrer l'instance au sein du JCRE en invoquant une des méthodes `register()`. La méthode `install()` prend un vecteur d'octets comme paramètre. Ce vecteur contient les paramètres d'installation pour l'initialisation et la personnalisation de l'instance d'applet.

Une applet Java Card reste inactive jusqu'à ce qu'elle soit explicitement sélectionnée. Lorsque le JCRE reçoit une commande `SELECT APDU`, il consulte sa table interne pour trouver l'applet dont l'AID correspond à celui spécifié dans la commande. S'il le trouve, le JCRE prépare la sélection de la nouvelle applet. Cette préparation se fait en deux étapes: d'abord, si une applet couramment sélectionnée est prése-

-nte en mémoire, alors le JCRE la désélectionne en invoquant la méthode `deselect()`. L'applet exécute la méthode deselect() avant de devenir inactive. Le JCRE invoque alors la méthode `select()` pour informer la nouvelle applet de sa sélection. La nouvelle applet effectue éventuellement une opération d'initialisation avant d'être réellement sélectionnée. L'applet retourne vrai à la méthode `select()` si elle est prête à devenir active et à traiter n'importe quelle commande APDU. Sinon, l'applet retourne faux pour décliner sa participation. La classe `javacard.framework.Applet` fournit une implémentation par défaut pour les méthodes `select()` et `deselect()`. Une sous-classe de la classe Applet peut redéfinir ces méthodes pour associer un autre comportement à ces méthodes.

Une fois l'applet sélectionnée, le JCRE fait suivre toutes les commandes APDU (y compris la commande `SELECT`) à la méthode `process()` de l'applet. Dans la méthode `process()`, l'applet interprète chaque commande APDU et exécute la tâche spécifiée par la commande. Pour chaque commande APDU, l'applet répond au CAD en envoyant une réponse APDU qui informe le CAD du résultat du traitement de la commande APDU. La méthode `process()` de la classe `javacard.framework.Applet` est une méthode de type `abstract`: une sous-classe de la classe `Applet` doit redéfinir cette méthode pour implémenter les fonctions de l'applet. Ce dialogue commande-réponse continue jusqu'à ce que une nouvelle applet soit sélectionnée ou bien que la carte soit retirée du CAD. Lorsqu'elle est désélectionnée, l'applet devient inactive jusqu'à sa prochaine sélection.

La méthode `getShareableInterfaceObject` sert dans la communication inter-applet. Elle est appelée par une applet cliente qui demande, à une applet serveur, à partager l'interface d'un objet. L'implémentation par défaut de cette méthode est de retourner `null`.

Étant donné que la commande APDU `SELECT` est aussi dirigée vers la méthode `process()`, la méthode `selectingApplet()` est utilisée par la méthode `process()` de l'applet pour distinguer entre la commande APDU `SELECT` qui sélectionne cette applet et les autres commandes APDU `SELECT` relatives à la sélection de fichiers ou de l'état interne de l'applet.

II.2.4.4 Définition de l'interface entre l'applet et le terminal :

Une applet qui tourne sur une carte à puce communique avec l'application en utilisant le protocole APDU (Application Protocol Data Units défini par l'ISO 7816). Par essence, l'interface entre l'applet et l'application est un ensemble de commandes APDU qui sont supportées aussi bien par l'applet que l'application.

La classe `javacard.framework.APDU` encapsule les commandes APDU. Elle fournit une interface puissante et flexible qui permet aux applets de gérer les commandes APDU. La classe APDU est conçue pour cacher les complexités du protocole, afin que les développeurs d'applet se concentrent davantage sur les détails de l'application.

Lorsque le JCRE reçoit une commande APDU, il encapsule la commande dans un objet APDU qu'il passe à la méthode `process()` de l'applet courante. L'objet APDU comporte un vecteur d'octets qui contient le message APDU. L'applet traite une commande APDU en invoquant des méthodes sur cet objet APDU. En général, l'applet effectue les étapes suivantes:

Étape 1. Extraire le buffer APDU:

L'applet invoque la méthode `getBuffer()` afin d'obtenir une référence au buffer APDU, qui contient le message. Lorsque l'applet reçoit l'objet APDU, seuls les 5 premiers octets sont disponibles dans le buffer. Il s'agit dans l'ordre des octets `CLA`, `INS`, `P1`, `P2`, et `P3`. L'octet `P3` désigne l'octet `Lc`, si la commande possède des données optionnelles. L'applet peut vérifier les octets entête pour déterminer la structure de la commande et l'instruction spécifiée par la commande.

Étape 2. Recevoir des données:

Si la commande APDU contient des données optionnelles, l'applet doit diriger l'objet APDU vers la réception de données entrantes en invoquant la méthode `setIncomingAndReceive()`. Les données sont lues dans le buffer APDU en suivant l'ordre des 5 octets d'entête. Le dernier octet de l'entête (`Lc`) indique la longueur des données entrantes. Si le buffer APDU ne peut contenir toutes les données, l'applet peut traiter les données en fragments, ou bien le copier vers un buffer interne. Dans les deux cas, elle doit faire appel de manière répétitive à la méthode `receiveBytes()` afin de lire les données additionnelles à partir du buffer APDU.

`arrayCopyNonAtomic(byte[] src, short srcOff, byte[] dest, short destOff,short length)` copie un vecteur à partir du vecteur source spécifié, en commençant de la position spécifiée à la position spécifiée dans le vecteur destination de manière non atomique.

Étape 3. Renvoyer des données:

Après avoir traité une commande APDU, l'applet peut retourner des données à l'application sous forme de réponses APDU. L'applet doit d'abord faire appel à la méthode `setOutgoing()` pour obtenir la longueur de la réponse (`Le`). `Le` est spécifié dans la commande APDU associée à la réponse APDU.

Ensuite, l'applet appelle la méthode `setOutgoingLength()` pour informer le CAD de la longueur réelle des données de la réponse. L'applet peut transférer les données vers le buffer APDU et appeler la méthode `sendBytes()` pour envoyer les données. La méthode `sendBytes()` peut être appelée plusieurs fois si le buffer APDU ne peut pas contenir toutes les données retournées.

Si les données sont stockées dans un buffer interne, l'applet invoque la méthode `sendByteLong()` pour envoyer les données à partir du buffer.

Si les données de la réponse sont trop courtes pour tenir dans le buffer APDU, la classe APDU fournit une méthode appropriée: `setOutgoingAndSend()`. Cette méthode est une combinaison de `setOutgoing`, de `setOutgoingLength` et de `sendBytes`. Néanmoins cette méthode ne peut être invoquée qu'une seule fois, et aucune méthode d'envoi ne peut être appelée après.

Étape 4. Renvoyer le mot d'état (word status):

Après un succès de la méthode `process()`, le JCRE envoie automatiquement 0x9000 pour indiquer un traitement normal. A n'importe quel point, si l'applet détecte une erreur, l'applet peut lever l'exception `ISOException` en appelant la méthode statique `ISOException.throwIt(short reason)`. Le mot d'état est spécifié dans le paramètre `reason`. Si l'exception `ISOException` n'est pas gérée par l'applet, elle sera attrapée par le JCRE. Le JCRE extrait le code de `reason` code et l'envoie comme mot d'état.

Remarque : Construction du code de l'applet.

Une fois la phase de conception de l'applet est finie, la seconde phase consiste à écrire le code de l'applet. L'utilisation de l'environnement de développement JBuilder/GemXplore nous permet de compiler l'applet en .class, de générer à partir de l'applet un fichier .cap qui sera le fichier à charger sur la carte, d'installer l'applet, de la sélectionner en vue de dialoguer avec elle en envoyant des commandes APDU. L'interface graphique étant très conviviale, il suffira de se laisser guider pour chaque étape. Notons par exemple que l'AID pour le package ainsi que celui de l'applet seront fournis à la création du projet de l'applet. De plus, l'envoi de commandes APDU pourra se faire directement en utilisant cette interface.

Lorsque l'utilisateur choisit de créer une applet Java Card, GemXplore/JBuilder crée automatiquement une applet contenant des méthodes prédéfinies :

- la méthode **`install()`** : permet l'installation de l'applet sur la carte en faisant appel au constructeur de l'applet.
- le **constructeur de l'applet** : doit initialiser des variables internes et appeler la méthode `register()` en vue de s'enregistrer auprès du JCRE.
- la méthode **`process()`** : sert à traiter toute commande APDU en vue de retourner une réponse.
- les méthodes **`select()`** et **`deselect()`** servent respectivement à la sélection de l'applet ou à sa désélection.

II.3 Les apports de la technologie Java Card : [16]

La technologie Java Card hérite de plusieurs avantages de sa technologie mère Java. Elle a largement été adoptée, et le déploiement de cartes à technologie Java Card a connu en 2004 une augmentation de 50%.

II.3.1 Flexibilité :

Java Card hérite de la méthodologie orientée objet du langage Java apportant ainsi toute sa souplesse dans le développement pour les cartes à puce et s'adressant à un nombre important de développeurs Java.

II.3.2 Compatibilité :

L'API Java Card est compatible avec les normes internationales comme ISO-7816, et avec les standards d'industrie comme EMV et GSM.

II.3.3 Portabilité du code et interopérabilité :

Les applets peuvent s'exécuter sur toute carte à puce. Cela permet d'utiliser une application sur toute puce compatible Java Card.

II.3.4 Multi-application :

C'est le fait d'autoriser la cohabitation de plusieurs applications sur la même carte.

II.3.5 Ajout de programmes après délivrance de la carte :

La possibilité d'ajouter les programmes permet de répondre aux évolutions des besoins du client. Il devient possible, par exemple, de changer l'algorithme d'authentification pour permettre une sécurité accrue sans changer de carte.

II.3.6 Environnement d'exécution sécurisé :

La technologie Java apporte plusieurs niveaux de contrôle d'accès aux méthodes et aux variables, un langage fortement typé, l'impossibilité de construire des « pointeurs », et la pare-feu des applets.

II.4 Les spécifications Java Card 3.0 :

Bien des raisons ont poussé à proposer une nouvelle version de Java Card : c'est la Java Card 3.0 qui existe en deux éditions distinctes : ce sont les éditions classique et connectée. Parmi ces raisons on peut citer : [31]

- les cartes à puce sont de plus en plus puissantes :
    - Processeurs 32 bits,
    - Mégaoctets de ROM et EPROM/Flash,
    - 16 ou 32 Kilooctets de RAM,
    - E/S haute vitesse (USB et autres) ;
- peine dans la programmation en Java Card 2 ;
- les fabricants de cartes à puce veulent plus de valeur ajoutée (la carte doit être en mesure d'abriter des services plus complexes) ;
- les opérateurs veulent des applications plus riches :
    - Serveurs Web sur cartes à puce,
    - Meilleure interaction avec les utilisateurs.

II.4.1 Java Card classique : [31], [w11]

La Java Card 3.0 classique est une évolution de la Java Card 2.2.2. La Java Card 3.0 est très similaire à la version 2.2.2 du point de vue langage (machine virtuelle et APIs 16 bits).

Elle cible les applications basées sur les APDU.

Elle est compatible en arrière, même au niveau binaire (les fichiers CAP existants peuvent être utilisés sans recompilation).

Elle n'apporte pas de réelles innovations mais procure de simples améliorations :

- support amélioré pour le sans contact, nouveau ISO7816-4 ;
- nouveaux algorithmes cryptographiques ;
- petits bugs fixés.

Les trois spécifications : Environnement d'Exécution, Machine Virtuelle et les API pour la plateforme Java Card, édition classique, peuvent être trouvées dans le site de SUN [w11].

II.4.2 Java Card connectée : [31], [w11]

La Java Card 3.0 connectée inclue de nouvelles caractéristiques (machine virtuelle 32 bits, multithreading, …etc).

Elle cible les applications web basées sur IP, ce qui est souhaitable pour les serveurs web pour cartes à puce.

Elle supporte les applications Java Card 2.2 (compatibilité arrière du code source).

Dans l'édition connectée, une nouvelle machine virtuelle et un nouvel environnement d'exécution sont introduits et ils supportent les trois modèles d'application contrairement au seul modèle d'application supporté par l'édition classique et les éditions précédentes de la plateforme Java Card.

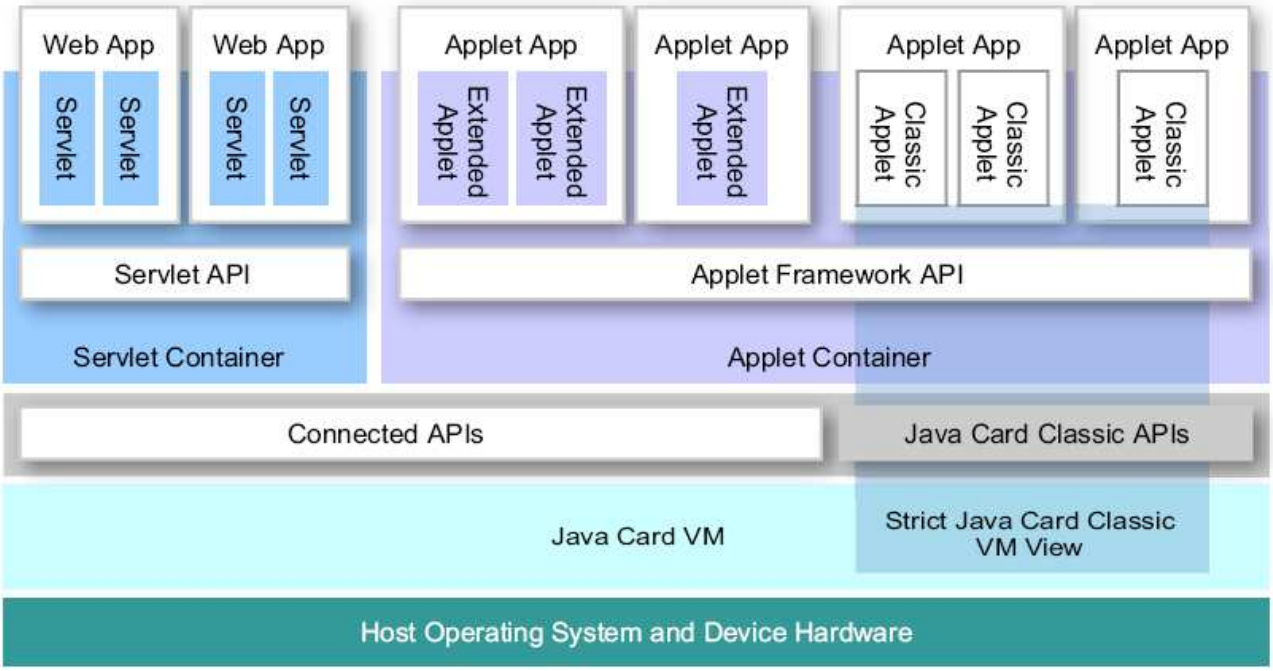
■ le modèle application applet classique : les applications basées sur les applets avec les mêmes capacités que les versions antérieures de la plateforme Java Card et dans l'édition classique. Ces applets utilisent le schéma basé sur les APDU pour la communication avec la carte.

■ le modèle application applet étendue : les applets ont des capacités plus avancées que les applets classiques. Ces applets utilisent le schéma basé sur les APDU pour la communication avec la carte.

■ le modèle application web : les applications sont basées sur les servlets qui utilisent le protocole http pour supporter un schéma basé sur le web pour la communication avec la carte.

Les trois spécifications : Environnement d'Exécution, Machine Virtuelle et les API pour la plateforme Java Card, édition connectée, peuvent être trouvées dans le site de SUN [w11].

Architecture de la Java Card 3 :



II.5 Conclusion :

Dans ce chapitre, nous avons essayé de détailler l'architecture de la technologie Java Card, qui offre un langage de programmation avec de bonnes propriétés, en donnant des informations sur l'installation et l'exécution d'une applet sur une carte à puce.

Le chapitre suivant sera consacré à la gestion des transactions.

CHAPITRE 3

Gestion des Transactions

Chapitre Trois :

Gestion des transactions

III.1 introduction :

L'utilisation de la notion de transaction a été initialement introduite dans les systèmes de gestion de base de données autour des années 1970 , et a été depuis généralisé à tous les systèmes d'informations. Cette notion joue désormais un rôle fondamental dans les applications traditionnelles de télécommunication, de contrôle de processus industriel, de finance, ou encore de gestion d'agences de voyages, que dans les nouvelles applications, telles que le commerce électronique sur le World Wide Web. Ces applications sont basées sur des systèmes d'informations complexes dans lesquels plusieurs programmes (ou utilisateurs) se partagent des informations à travers des réseaux. Dans ce contexte, la notion de transaction répond au besoin fondamental de simplifier le problème du maintien de la cohérence des informations en cas d'accès concurrents ou de défaillances.

III.2 Le modèle transactionnel : [17], [8], [3], [w9]

Les cartes à puce s'ouvrent de plus en plus aux systèmes distribués. Nous nous retrouvons donc dans un contexte d'exécution distribué et sujet aux fautes (encombrement du réseau, non disponibilité du serveur, ou non présence de la carte à puce). Le traitement de ces fautes par le programmeur se révèle comme étant un véritable casse-tête.

Une transaction peut être définie comme étant une suite d'actions commençant par un « Début_Transaction » (Begin_ Transaction en anglais) et se terminant par un « Validation_Transaction » (Commit_Transaction en anglais) si tout se passe bien, ou par un « Annulation_Transaction » (Rollback ou Abort en Anglais) s'il y a eu un problème.

Le modèle transactionnel définit un modèle d'exécution pour applications simultanées, qui garantissent un certain nombre de propriétés : ce sont les propriétés ACID[1].

III.2.1 Les propriétés ACID : [17]

Pour illustrer les propos sur les propriétés des transactions, nous allons utiliser un exemple très simple de transaction : le débit d'un compte A pour créditer un compte B :

1. ACID : Atomicité, Cohérence, Isolation et Durabilité

```
BeginTransaction
  CompteA.débit(100) ;
  CompteB.crédit(100) ;
CommitTransaction
```

III.2.1.1 Atomicité :

L'atomicité d'une transaction peut être définie comme étant la règle de « tout ou rien ». En effet, lorsqu'on exécute une transaction, on doit être en mesure d'assurer que, soit que toutes les actions de la transaction ont été validées, soit aucune de ces actions n'est conservée. Si nous prenons l'exemple de notre virement bancaire cela signifie que s'il y a une panne après le débit du compte A, ce compte doit être recrédité, car la transaction n'a pas été validée.

En pratique, cela signifie qu'aucun effet d'une transaction annulée ne doit apparaître dans le système. L'atomicité est du ressort de la tolérance aux pannes.

III.2.1.2 Cohérence :

La propriété de maintien de la cohérence des données dit qu'une transaction doit prendre des données dans un état cohérent et de les rendre dans un autre état cohérent. La cohérence est obligatoirement violée pendant la transaction. Si on prend l'exemple de la transaction du versement bancaire, la cohérence des données n'est pas respectée après le débit (et avant le crédit), mais tout rentre dans l'ordre à la fin de la transaction.

Le maintien de la propriété de cohérence est donc principalement du ressort du programmeur. La propriété de la cohérence veille à ce que les règles de travail sur les données définies par le programmeur, soient respectées, en garantissant l'exécution de toutes les actions prévues par le programmeur.

III.2.1.3 Isolation :

Cette propriété dit que les modifications d'une transaction sont invisibles aux transactions concurrentes. Ce qui signifie, que lors de l'exécution concurrente de plusieurs transactions, chaque transaction doit apparaître comme si elle s'exécutait seule.

Prenons comme exemple la figure III.a (page suivante). Dans le premier cas, les deux transactions s'exécutent simultanément, mais le résultat final (c'est-à-dire après les deux transactions) ne serait pas le même si ces transactions s'étaient exécutées individuellement. Il faut donc utiliser des mécanismes de ***sérialisation***[2], qui ordonnent les actions des transactions, pour respecter la propriété de l'isolation (exemple (2) dans la figure III.a).

2. Une exécution de transactions est sérialisable si ses effets sont les mêmes qu'en exécutant les transactions les unes après les autres

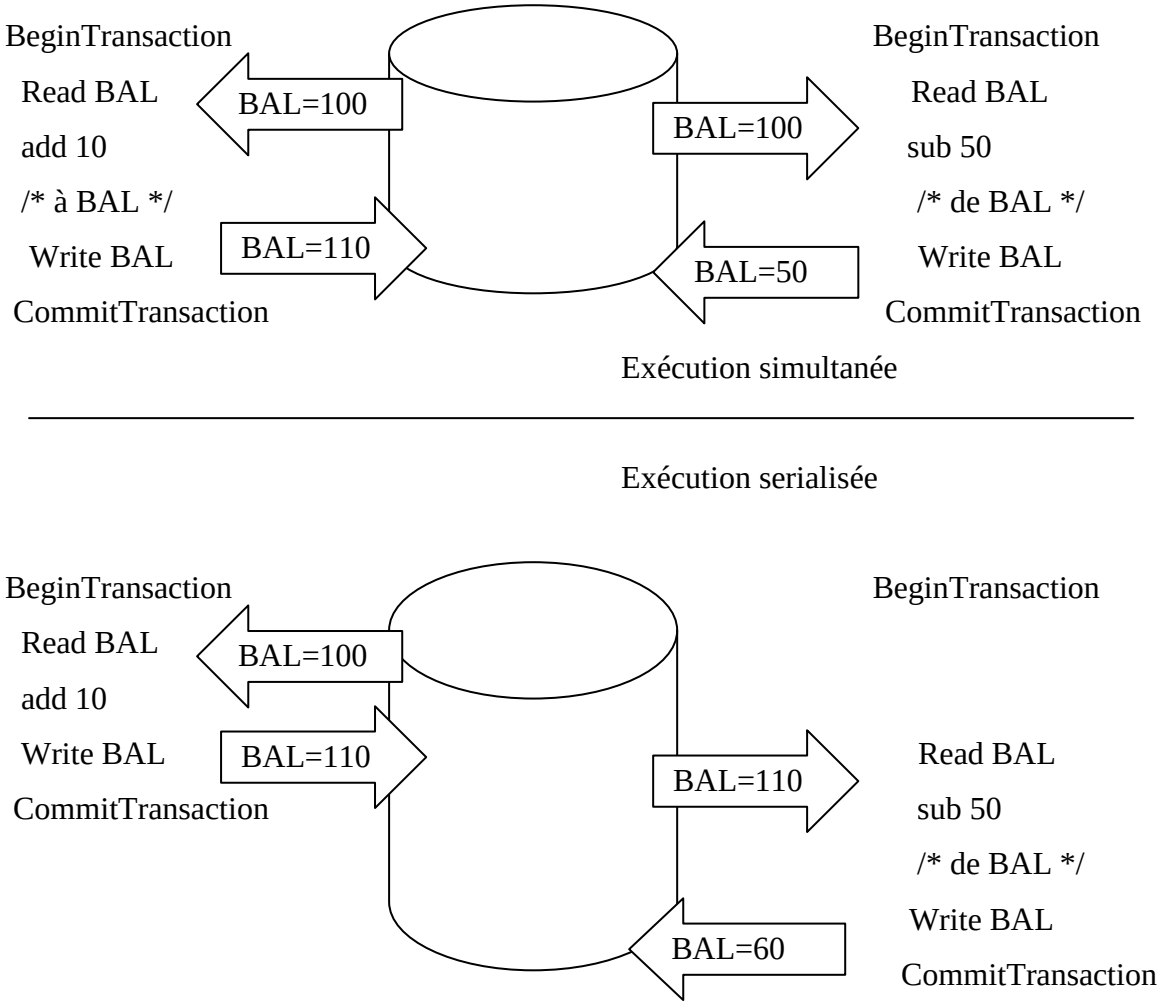


Fig III.a : *La propriété de l'isolation*

III.2.1.4 Durabilité :

La propriété de la durabilité implique que les actions d'une transaction ne peuvent pas être perdues si la transaction est validée. Cela signifie que les résultats d'une transaction doivent être conservés, même après une panne.

Il est évident que la durabilité absolue n'existe pas, mais on peut définir plusieurs niveaux de protection des résultats (possibilité de refaire les actions perdues, ou support de sauvegarde).

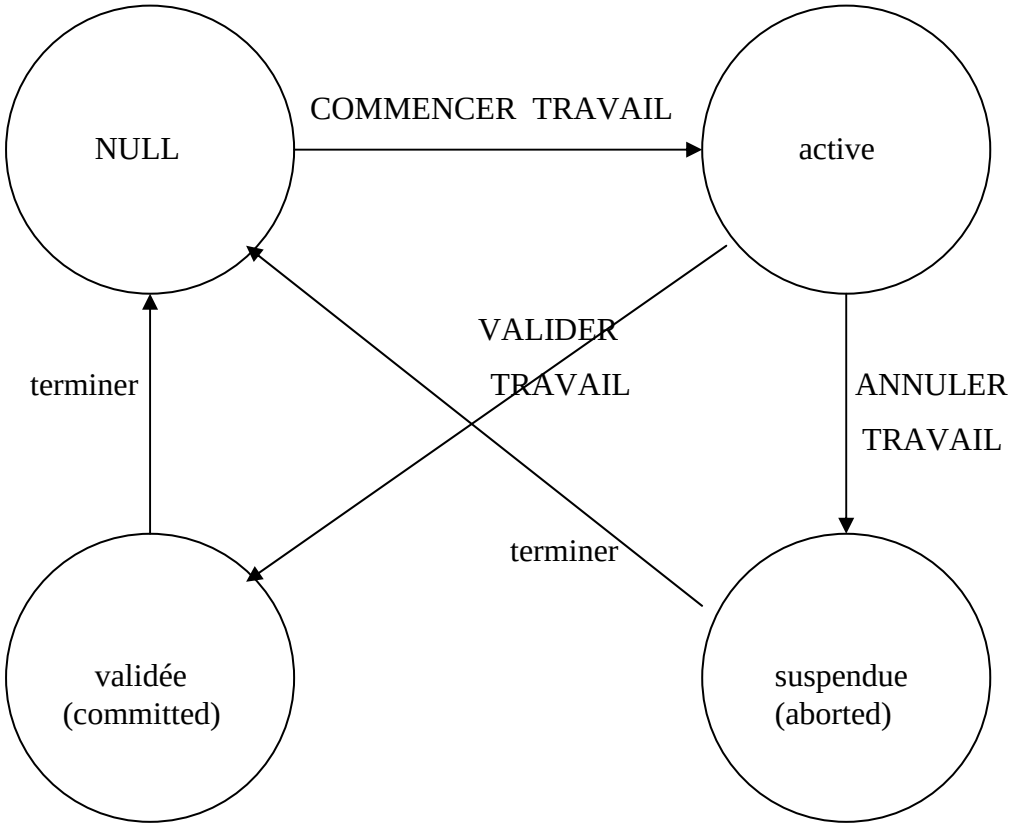
La durabilité est du ressort des mécanismes de reprise sur panne.

Remarque :

Comme nous avons pu le voir au cours de l'explication des diverses propriétés, seuls deux mécanismes sont nécessaires pour implanter les propriétés ACID dans un système : il faut un mécanisme de sérialisation et un mécanisme de reprise sur panne [17].

III.2.2 Etats d'une transaction : [12]

Nous pouvons schématiser les états d'une transaction comme suit :



III.2.3 Théorie de la concurrence : [8]

Cette section aborde le problème de gestion des accès concurrents. Les solutions proposés permettent de garantir la cohérence et l'isolation des mises à jour des transactions (le C et le I de ACID). Elles sont basées sur la théorie de la sérialisabilité des transactions, que nous examinons maintenant.

Mais observons d'abord les deux lois du contrôle de concurrence :

- Première loi du contrôle de concurrence :

  l'exécution concurrente de doit pas causer le disfonctionnement des programmes d'application.

- Seconde loi du contrôle de concurrence :

  l'exécution concurrente ne doit pas avoir un débit plus bas ou un temps de réponse plus haut qu'une exécution série.

III.2.3.1 Objectifs :

L'objectif général est de rendre invisible aux clients le partage simultané des données. Cette transparence nécessite des contrôle des accès concurrents au sein du système transactionnel. Ceux-ci s'effectuent au moyen de protocoles spéciaux permettant de synchroniser les mises à jour afin d'éviter les pertes de mises à jour et l'apparition d'incohérences.

Une perte de mise à jour survient lorsqu'une transaction T1 exécute une mise à jour calculée à partir d'une valeur périmée de donnée, c'est-à-dire d'une valeur modifiée par une autre transaction T2 depuis la lecture par la transaction T1. La mise à jour de T2 est donc écrasée par celle de T1. Une perte de mise à jour est illustrée par la figure III.b : la mise à jour de la transaction T2 est perdue.

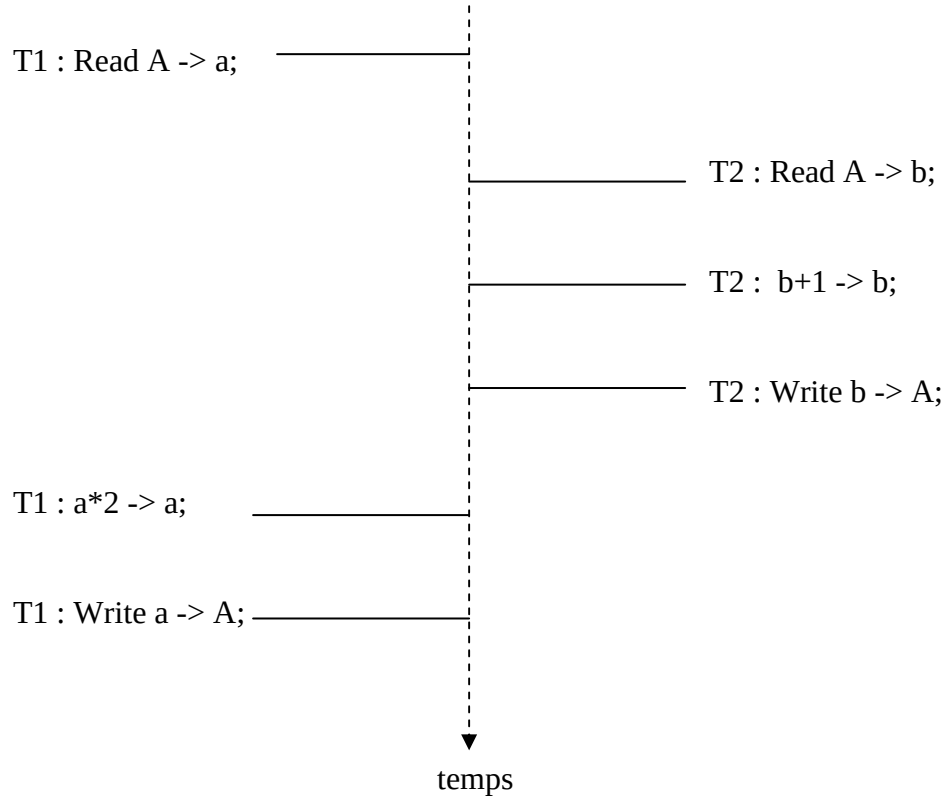


Figure III.b : *Exemple de perte de mise à jour*

Une incohérence apparaît lorsque des données liées par une contrainte d'intégrité sont mises à jour par deux transactions dans des ordres différents, de sorte à enfreindre la contrainte. Par exemple, soient deux données A et B devant rester égales. L'exécution de la séquence d'opérations suivante :

{ T1 : A = A+1 ;
T2 : B = B+2 ;
T2 : A = A*2 ;
T1 : B = B+1 }

rend en général A différent de B du fait de la non-commutativité de l'addition et de la multiplication. Elle provoque donc l'apparition d'une incohérence.

Cette situation est illustrée dans la figure III.c.

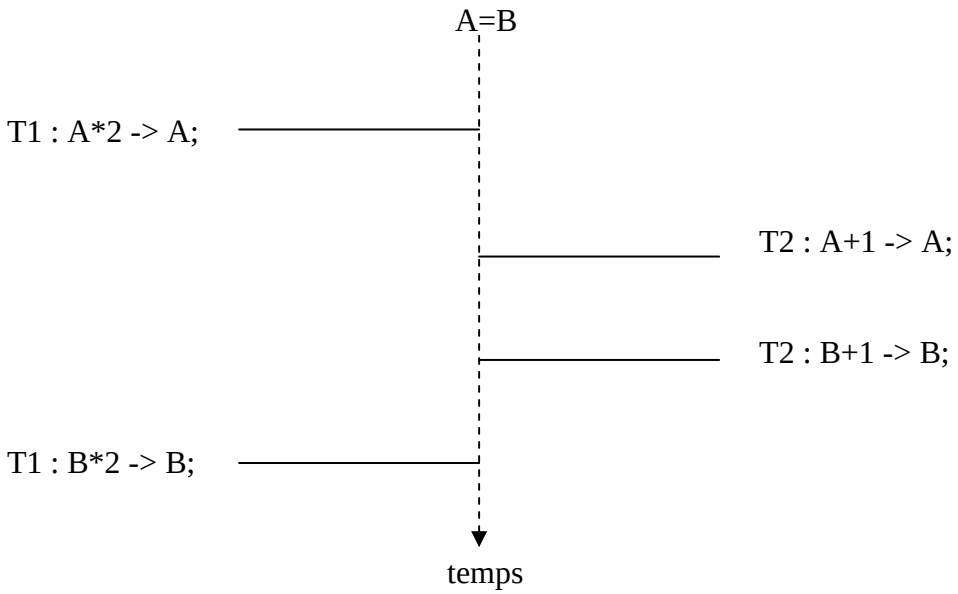


Figure III.c : *Exemple d'introduction d'incohérence*

Un autre problème lié aux accès concurrents est la non-reproductibilité des lectures : deux lectures d'une même donnée dans une même transaction peuvent conduire à des valeurs différentes si la donnée est modifiée si la donnée est modifiée par une autre transaction entre les deux lectures (voir figure III.d). Le problème ne survient pas si les mises à jour sont isolées, c'est-à-dire non visibles par une autre transaction avant la fin de la transaction. Il n va de même de l'apparition d'incohérences. Pour les pertes de mise à jour, l'isolation des mises à jour n'est pas suffisante : il faut aussi ne pas laisser deux transactions modifier simultanément la même donnée.

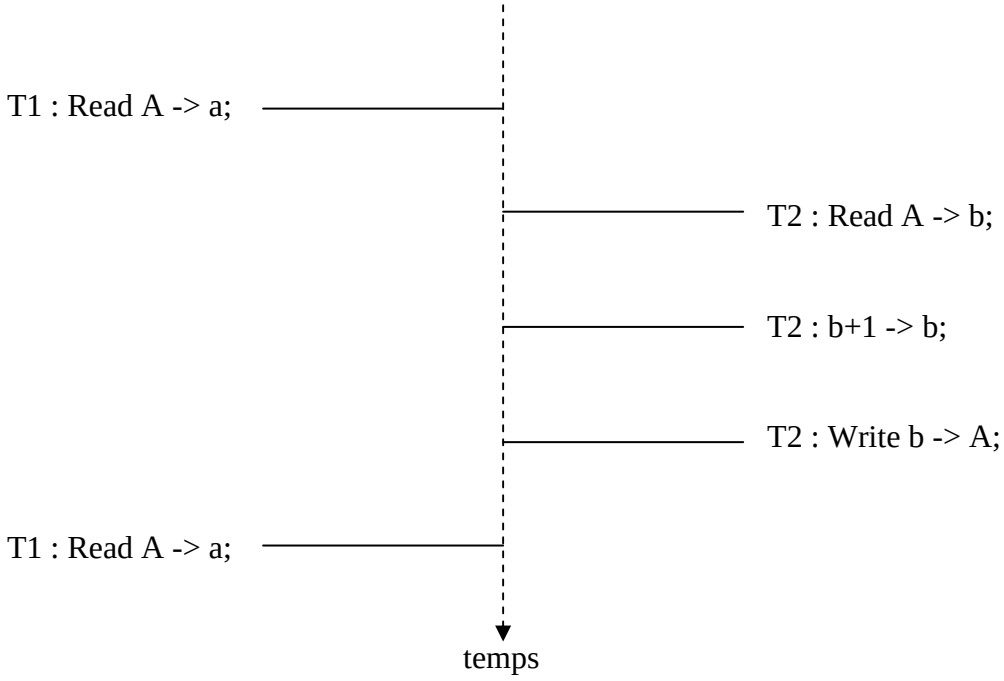


Figure III.d : *Exemple de non-reproductibilité des lectures*

La résolution dans un système des problèmes évoqués nécessite la mis en place d'algorithmes de contrôle de concurrence spécialisés. Ces algorithmes s'appuient sur la théorie de la concurrence que nous examinons ci-dessous.

III.2.3.2 Quelques définitions de base : [8]

Pour éviter perte d'opérations, incohérences et non-reproductibilité des lectures, le système doit contrôler l'accès aux données. L'unité de données contrôlée dépend du système. De plus en plus de systèmes permettent des contrôles variables selon le type des transactions. Nous appellerons cette unité granule de concurrence ; le terme objet est parfois aussi employé.

Définition : **Granule de concurrence** (*Concurrency granule*)

Unité de données dont les accès sont contrôlés individuellement par le système.

Dans un SGBD relationnel un granule au sens de la concurrence peut être une ligne, une page ou une table. Ce put être un objet ou une page dans un SGBD objet.

Les granules de concurrence sont lus et écrits par les utilisateurs, éventuellement par parties. On appelle Action un accès élémentaire à un granule.

Définition : **Action** (*Action*)

Unité indivisible exécutée par le système sur un granule pour un utilisateur, constituée généralement par une lecture ou une écriture.

Un système transactionnel exécute donc une suite d'actions résultant de transactions concurrentes. Après complétude d'un ensemble de transactions (T1, T2, … Tn), une histoire du système peut être représentée par la suite des actions exécutées. Plus généralement, toute suite d'actions pouvant représenter une histoire possible sera appelée simplement exécution.

Définition : **Exécution de transactions** (*Schedule*, ou *Log*, ou *History*)

Séquence d'actions obtenues en intercalant les diverses actions des transactions T1, T2, … Tn tout en respectant l'ordre interne des actions de chaque transaction.

Une exécution respecte donc l'ordre des actions de chaque transaction participante et est, par définition, séquentielle. Par exemple, considérons les transactions T1 et T2 de la figure III.e, modifiant les données A et B reliées par la contrainte d'intégrité A=B ; A et B appartenant à des granules distincts, maximisant ainsi les possibilités de concurrence. Une exécution correcte de ces deux transactions est représentée en figure III.f (a). Une autre exécution est représentée en figure III.f (b), mais celle-là est inacceptable car elle conduit à une perte d'opérations.

T1	**T2**
Read A → a1	Read A → a2
a1 + 1 → a1	a2 * 2 → a2
Write a1 → A	Write a2 → A
Read B → b1	Read B → b2
b1 + 1 → b1	b2 * 2 → b2
Write b1 → B	Write b2 → B

Figure III.e : *Deux transactions T1 et T2*

T1: Read A → a1	T2: Read A → a2
T1: a1 + 1 → a1	T2: a2*2 → a2
T1: Write a1 → A	T1: Read A → a1
T2: Read A → a2	T1: a1 + 1 → a1
T2: a2 * 2 → a2	T2: Write a2 → A
T2: Write a2 → A	T2: Read B → b2
T1: Read B → b1	T2: b2 * 2 → b2
T1: b1 + 1 → b1	T1: Write a1 → A
T1: Write b1 → B	T1: Read B → b1
T2: Read B → b2	T1: b1 + 1 → b1
T2: b2 * 2 → b2	T1: Write b1 → B
T2: Write b2 → B	T2: Write b2 → B
(a)	(b)

Figure III.f : *Deux exécutions des transactions T1 et T2*

III.2.3.3 Propriétés des opérations sur granule : [8]

Un granule accédé concurremment obéit à des contraintes d'intégrité internes. Lors de la modification des données, les granules sont modifiés par des suites d'actions constituant des unités fonctionnelles appelés opérations. Les opérations respectent la cohérence interne du granule, c'est-à-dire les contraintes d'intégrité qui relient les données appartenant au granule.

Définition : **Opération** (*Operation*)

Suite d'actions accomplissant une fonction sur un granule en respectant sa cohérence interne.

Par exemple si le granule est la page dans un SGBD, les opérations de base sont souvent LIRE(page) et ECRIRE(page), qui sont également dans bien des systèmes des actions indivisibles. Si le granule est l'article, des opérations plus globales nécessitant plusieurs actions indivisibles sont LIRE(article) et ECRIRE(article), mais aussi MODIFIER(article) et INSERER(article).

Avec ces opérations de base, il est possible d'en construire d'autres plus globales encore. Sur un objet typé, tel un compte en banque, on peut distinguer des opérations, créer, créditer, débiter, détruire, etc…

L'application d'opérations à des granules conduit à des résultats. Le résultat d'une opération est constitué par l'état du granule concerné après l'application de l'opération considérée et par les effets de bord qu'elle provoque. Par exemple, le résultat d'une opération LIRE est représenté par la valeur du tampon récepteur près exécution, alors que le résultat d'une transaction modifiant une base de données est l'état des granules modifiés après exécution ainsi que la valeur des messages édités.

Les opérations sont enchevêtrées au niveau des actions lors de l'exécution simultanée de transactions. Deux opérations qui ne modifient aucun granule et qui appartiennent à deux transactions différentes peuvent être enchevêtrées de manière quelconque sans modifier le résultat de leur exécution. Autrement dit, toute intercalation d'opérations n'effectuant que des lectures conduit à des résultats identiques à une exécution successive de ces opérations. Plus généralement, il est possible de définir la notion d'opérations compatibles.

Définition : **Opérations compatibles** (*Compatible operation*)

> Opérations Oi et Oj dont toute exécution simultanée donne le même résultat qu'une exécution séquentielle Oi suivie de Oj ou de Oj suivie de Oi (à noter que les résultats Oi puis Oj t Oj puis Oi peuvent être différents).

Considérons par exemple les opérations représentées en figure III.g. Les opérations O11 et O21 sont compatibles ; O11 et O12 ne le sont pas.

Il est important de remarquer que deux opérations travaillent sur deux granules différents sont toujours compatibles. En effet, dans ce cas aucune perte d'opérations ne peut survenir si l'on intercale les opérations. Or il est simple de voir que deux opérations sont incompatibles lorsqu'il existe une possibilité d'intercalation générant une perte d'opérations.

O11

```
{ Read A → a1
  a1 + 1 → a1
  Write a1 → A }
```

O12

```
{ Read A → a2
  a2*2 → a2
  Write a2 → A }
```

O21

```
{ Read B → b1
  b1 + 1 → b1
  Write b1 → B }
```

O22

```
{ Read B → b2
  b2 * 2 → b2
  Write b2 → B }
```

(Figure III.g : *première partie*)

```
O31
{ Read A → a1
  a1 + 10 → a1
  Write a1 → A }
```

Figure III.g : *Exemple d'opérations*

Les problèmes surviennent avec les opérations incompatibles. Lorsqu'une au moins modifie un granule auquel l'autre a accédé. L'ordre d'exécution des deux opérations peut alors changer les résultats. Dans d'autres cas, il peut être indifférent. Plus généralement, nous définirons la notion d'opérations permutables qu'il faut bien distinguer de celle d'opérations compatibles (la première est une notion indépendante de l'ordre d'exécution, alors que la seconde est définie à partir de la comparaison des ordres d'exécution).

Définition : **Opérations permutables** (*Permutable operations*)

Opérations Oi et Oj telles que toute exécution de Oi suivie par Oj donne le même résultat que celle de Oj suivie par Oi.

Par exemple, les opérations O11 et O31 représentées en figure III.g sont permutables alors que les opérations O11 et O12 ne le sont pas. Soulignons que deux opérations qui travaillent sur des granules différents sont toujours permutables. En effet, dans ce cas, l'exécution d la première ne peut modifier le résultat de la seconde et réciproquement. Plus généralement, deux opérations compatibles sont permutables, mais la réciproque n'est pas vraie.

III.2.3.4 Caractérisation des exécutions correctes : [8]

Certaines exécutions introduisent des pertes d'opérations ou des inconsistances, comme nous l'avons vu ci-dessus. L'objectif du contrôle de concurrence consiste à ne laisser s'exécuter que des exécutions sans perte d'opérations ou inconsistances. Il est bien connu que l'exécution successive de transactions (sans simultanéité entre transactions) est un cas particulier d'exécution sans perte d'opérations ni inconsistances. Une telle exécution est appelée succession et peut être définie plus formellement comme suit :

Définition : **Succession** (*serial schedule*)

Exécution E d'un ensemble de transactions {T1, T2, … Tn} telle qu'il existe une permutation π de (1, 2 … n) telle que

$$E = < T\pi(1), \pi(2), \ldots T\pi(n) >$$

Afin d'assurer l'absence de conflits, il est simple de ne tolérer que les exécutions qui donnent le même résultat qu'une succession pour chaque transaction. De telles exécutions sont dites sérialisables.

Définition : **Exécution sérialisable** (*Seriable Schedule*)

Exécution E d'un ensemble de transactions (T1, T2, … Tn) donnant globalement le même résultat qu'une succession de (T1, T2, … Tn).

Le problème du contrôle de concurrence est donc de s'assurer qu'un système centralisé (ou réparti) ne peut générer que des exécutions sérialisables. C'est là une condition suffisante pour assurer l'absence de conflits dont la nécessité peut être discutée. En fait, la condition est nécessaire si le système n'a pas de connaissance sur la sémantique des opérations.

Afin de caractériser les exécutions sérialisables, nous introduisons deux transformations de base d'une exécution de transactions. Tout d'abord la séparation d'opérations compatibles Oi et Oj exécutées par des transactions différentes consiste à remplacer une exécution simultanée des opérations E (Oi,Oj) par la séquence donnant le même résultat, soit <Oi, Oj> ou <Oj, Oi>. La séparation d'opérations permet donc de mettre en succession des opérations compatibles exécutées par des transactions différentes. Ensuite, la permutation d'opérations permutables Oi et Oj exécutées par des transactions différentes consiste à changer l'ordre d'exécution de ces opérations ; par exemple la séquence <Oi ; Oj> est remplacée par <Oj ; Oi>.

Une condition suffisante pour qu'une exécution soit sérialisable est qu'elle puisse être transformée par séparation des opérations compatibles et permutations des opérations permutables en une succession des transactions composantes. En effet, par définition, séparations et permutations conservent les résultats. Par suite, si l'exécution peut être transformée en une succession, elle donne le même résultat que cette succession pour chaque transaction et est donc sérialisable. La condition n'est pas nécessaire car, au moins pour certaines valeurs des données, ds opérations incompatibles ou non permutables peuvent être exécutées simultanément sans conflits.

A titre d'exemple considérons l'exécution représentée en figure III.f (a). En représentant seulement globalement les opérations, cette exécution s'écrit :

T1: $A + 1 \rightarrow A$

T2: $A * 2 \rightarrow A$

T1: $B + 1 \rightarrow B$

T2: $B * 2 \rightarrow B$

Les opérations $A * 2 \rightarrow A$ et $B + 1 \rightarrow B$ sont permutables car elles agissent sur des granules différents.

Par suite, cette exécution peut être transformée en :

T1: A + 1 → A

T1: B + 1 → B

T2: A * 2 → A

T2: B * 2 → B

qui est une succession de T1 puis T2. Par suite, l'exécution de la figure III.f (a) est sérialisable.

III.2.3.5 Graphe de précédence : [8]

Une exécution sérialisable est correcte car elle donne un résultat que l'on obtiendrait en exécutant les transactions l'une après l'autre. Lorsqu'on examine une séquence d'opérations résultant d'une exécution simultanée d'un ensemble de transactions, il apparaît que l'ordre de certaines opérations ne peut être changé sans changer le résultat, du fait de la non-commutativité des opérateurs exécutés (par exemple, addition et multiplication).

Les chercheurs ont ainsi abouti à définir le notion de précédence de transactions dans une exécution simultanée.

Définition : **Précédence** (*Precedence*)

Propriété stipulant qu'une transaction a accompli une opération Oi sur une donnée avant qu'une autre transaction accomplisse une opération Oj, Oi et Oj n'étant pas commutatives ($\{Oi ; Oj\} \neq \{Oj ; Oi\}$).

La notion de précédence est générale et s'applique à tout type d'opération. En pratique, les systèmes ne considèrent d'ordinaire que les opérations de lecture et d'écriture. Les précédences sont alors créées par les séquences d'actions de base lecture et écriture. Les séquences non commutatives lecture puis écriture, écriture puis lecture, écriture puis écriture, d'une même donnée introduisent des précédences.

Plus précisément, l'une des séquences :

- Ti: lire(D) … Tj: écrire(D)
- Ti: écrire(D) … Tj: écrire(D)
- Ti: écrire(D) … Tj: lire(D)

implique que Ti précède Tj.

Considérons une exécution simultanée de transactions. La relation de précédence entre transactions peut être représentée par un graphe :

Définition : **Graphe de précédence** (*Precedency graph*)

Graphe dont les nœuds représentent les transactions et dans lequel il existe un arc de Ti vers Tj si Ti précède Tj dans l'exécution analysée.

Une exécution simultanée des transactions T1, T2 et T3 et le graphe de précédence associé sont illustrés en figure III.h.

Il est simple de montrer qu'une condition suffisante de sérialisabilité est que le graphe de précédence soit sans circuit. En effet, dans ce cas, il est toujours possible de transformer l'exécution simultanée en une succession en séparant puis permutant les opérations. L'ordre des transactions dans la succession et induit par le graphe sans circuit.

Par exemple, l'exécution simultanée représentée en figure III.h n'est pas sérialisable puisque le graphe de précédence possède un circuit.

{ T1: Lire A
T2: Ecrire A
T2: Lire B
T2: Ecrire B
T3: Lire A
T1: Ecrire B }

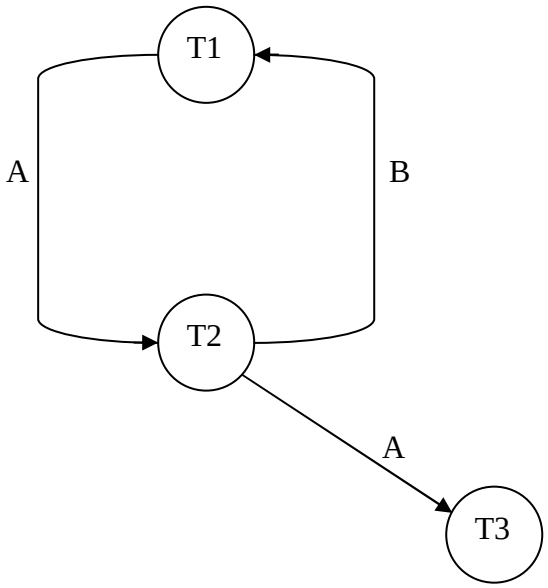


Figure III.h : *Exemple de graphe de précédence*

Remarque : D'une façon générale le graphe de dépendances n'est pas géré par les systèmes transactionnels, mais il existe deux approches utilisées comme heuristiques pour contrôler la sérialisabilité des transactions : les méthodes pessimistes (ce sont des techniques de prévention des conflits qui empêchent leur apparition) et les méthodes optimistes (ce sont des techniques de détection qui laissent les conflits se produire mais les détectent et annulent leurs effets).

III.2.4 Contrôle de concurrence pessimiste : [8]

Les méthodes pessimistes dites ainsi car elles préviennent des conflits qui ne surviennent en général pas. Elles sont basées sur le verrouillage.

III.2.4.1 Le Verrouillage deux phases : [8], [17], [3]

Le verrouillage deux phases est une technique de prévention des conflits basée sur le blocage des objets par des verrous en lecture ou en écriture avant d'effectuer une opération de sélection ou de mise à jour. En théorie, une transaction ne peut relâcher les verrous avant d'avoir obtenu tous ceux qui lui sont nécessaires, afin de garantir la correction du mécanisme.

Définition : **Verrouillage Deux Phases** (*Two Phase Locking – 2PL*)

> Technique de contrôle des accès concurrents consistant à verrouiller les objets au fur et mesure des accès par une transaction et à relâcher les verrous seulement après obtention de tous les verrous.

La mise en œuvre de 2PL repose sur l'utilisation de verrous. A chaque objet G sont associés deux types de verrous , les verrous en mode partagé (*shared locks* ou *S locks* en anglais), également appelés verrous de lecture (*read locks*) , et les verrous en mode exclusif (*exclusive locks* ou *X locks*) , ou verrous d'écriture (*write locks*). Deux opérations sont définies :

- l'acquisition des verrous dans le mode de lecture ou écriture , notée `Lock`,
- la libération du verrou notée `Unlock`.

L'obtention d'un verrou par une transaction dépend de la compatibilité de mode entre le verrou demandé et les verrous détenus par d'autres transactions. Si le mode demandé est compatible avec les modes de verrous détenus , la transaction obtient immédiatement le verrou et poursuit son exécution. Dans le cas contraire , la transaction est mise en attente jusqu'à la libération de verrou par d'autres transactions permette de satisfaire sa demande.

La compatibilité des verrous en modes lecture et écriture est donnée par la figure III.i.

	Lecture	Ecriture
Lecture	Oui	Non
Ecriture	Non	Non

Figure III.i : *compatibilité des verrous*

La règle d'utilisation des verrous par les transactions est la suivante : chaque opération de lecture (respectivement écriture) doit être précédée d'une demande de verrou en mode lecture (resp. écriture) et doit être suivie d'une libération de verrou. Cependant dès qu'une transaction a libéré un verrou il lui est interdit d'en acquérir de nouveaux. Une transaction est donc structurée en deux phases : une phase d'acquisition de verrous et une phase de libération de verrous. On appelle point de verrouillage maximum le point situé après l'acquisition du dernier verrou (φ dans la figure III.j).

Théorème :

Si chacune des transactions est à deux phases , une exécution concurrente de transactions validées est sérialisable [3].

Remarque :

L'ordre de sérialisation correspond à l'ordre de franchissement du point de verrouillage maximum.

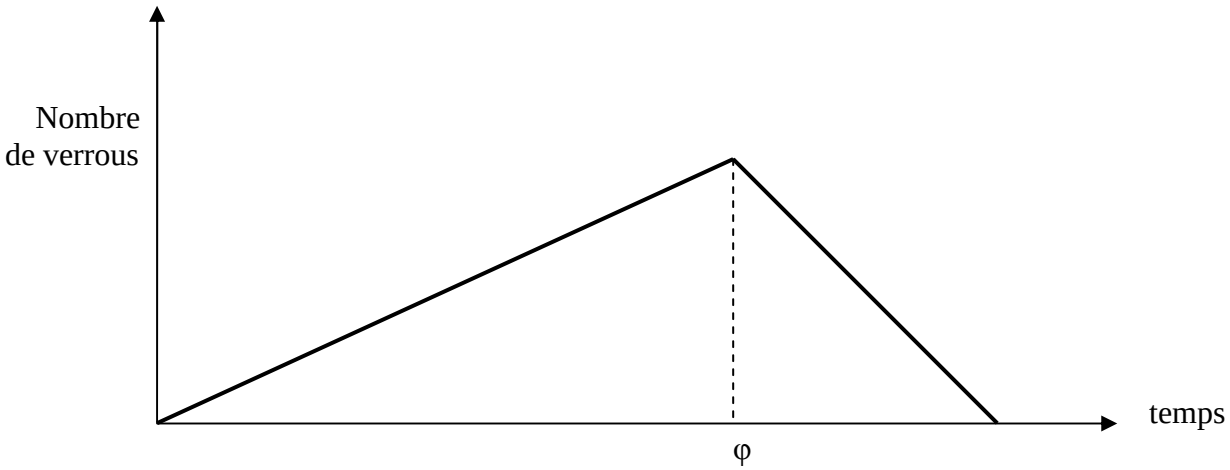


Figure III.j : *comportement des transactions deux phases* [8]

La figure III.j décrit les deux phases que comporte une transaction : une phase d'acquisition de verrous et une phase de relâchement. Cette condition garantit un ordre identique des transactions sur les objets accédés en mode incompatible. Cet ordre est celui d'exécution des points de verrouillage maximal φ.

Remarque :

En réalité le nombre de verrous évolue par une fonction en escalier :

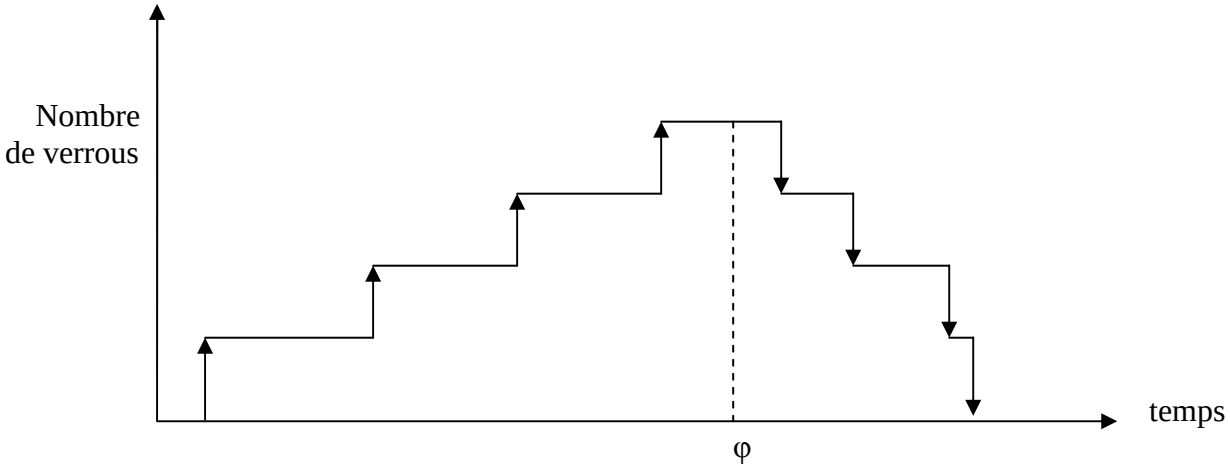


Figure III.j : *comportement réel des transactions deux phases* [19]

Dans la figure III.j, chaque côté montant est une action `Lock`, et chaque côté descendant est une action `Unlock`.

Exemple d'une transaction à deux phases :

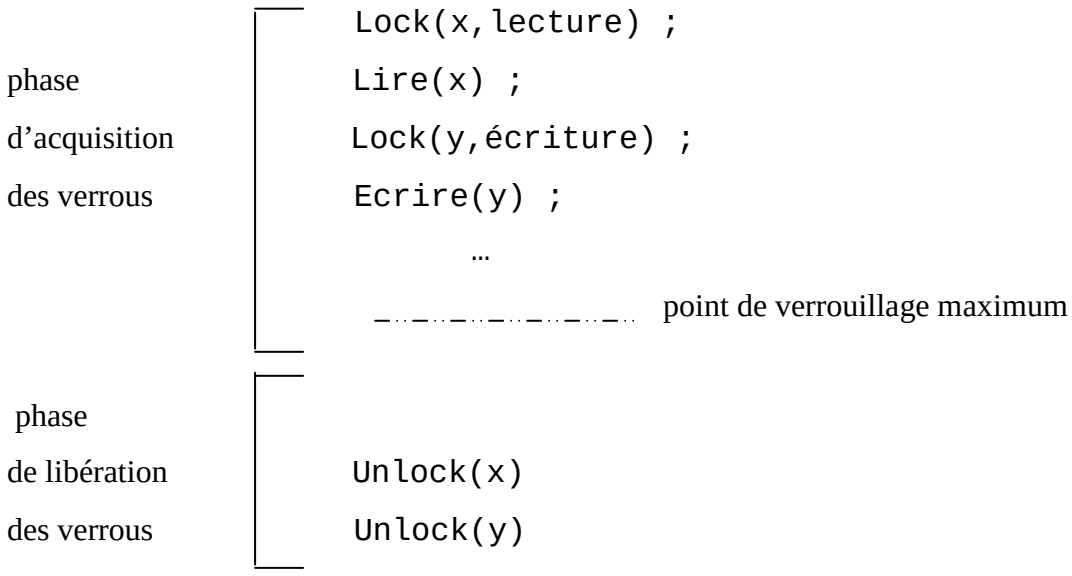


Figure III.l : *transaction à deux phases* [3]

Remarque :

En pratique, afin de garantir l'isolation des mises à jour , les verrous sont seulement relâchés en fin de transaction, lors de la validation.

Les algorithmes de verrouillage et déverrouillage sont détaillés dans ce qui suit : [8]

```
Fonction Lock(t :transaction, G :granule, M :mode) : booléen
Début
  Cverrou := 0
  Pour chaque transaction i≠t ayant verrouillé G faire
    Cverrou := Cverrou ∪ t.verrou(G) /* cumuler verrous sur G */
  FinPour
  Si compatible (Mode,Cverrou) alors
    t.verrou(G) := t.verrou(G) ∪ M /* marquer G verrouillé */
    Lock := TRUE
  Sinon
    Insérer (t,M) dans queue de G /* mise en attente de G */
    Bloquer la transaction t
    Lock := FALSE
  FinSi
Fin /* de la fonction Lock */
```

```
Procédure Unlock(t :transaction, G :granule)
Début
  t.verrou(G) := 0 /* remise à 0 du verrou de la transaction sur G */
  Pour chaque transaction i de la queue de G faire
    Si Lock(i,G,M) alors
      Enlever(i,M) de la queue de G
      Débloquer i
    FinSi
  FinPour
Fin /* de la procédure Unlock */
```

Figure III.m : *Algorithmes de verrouillage et déverrouillage*

L'application du verrouillage dans un système pose le problème du choix du granule de verrouillage. Dans une base de données relationnelle, les objets à verrouiller peuvent être des tables, des pages ou des tuples. Une granularité variable des verrous est souhaitable, les transactions manipulant beaucoup de tuples pouvant verrouiller au niveau table ou page, celles accédant ponctuellement à quelques tuples ayant la capacité de verrouiller au niveau du tuple. Le choix d'une unité de verrouillage fine (par exemple le tuple) minimise bien sûr les risques de conflits. Elle maximise cependant la complexité et le coût du verrouillage.

Remarque : Degré d'isolation en SQL2

Le verrouillage, tel que présenté ci-dessus, est très limitatif du point de vue des exécutions simultanées possibles. Afin de proposer une approche plus permissive et de laisser s'exécuter simultanément des transactions présentant des dangers limités de corruption des données, le groupe de normalisation de SQL2 a défini des degrés d'isolation emboîtés, du moins contraignant au plus contraignant, ce dernier correspondant au verrouillage deux phases. Le groupe distingue les verrous courts relâchés après exécution de l'opération et les verrous longs relâchés en fin de transaction. Le degré de verrouillage souhaité est choisi par le développeur de la transaction parmi les suivants :

- le degré 0 garantit les non pertes de mise à jour ; il correspond à la pose de verrous courts exclusifs lors des écritures.
- le degré 1 garantit la cohérence des mises à jours ; il génère la pose de verrous longs exclusifs en écriture par le système.
- le degré 2 assure la cohérence des lectures individuelles ; il ajoute la pose de verrous courts partagés en lecture à ceux de degré 1.

- le degré 3 atteste de la reproductibilité des lectures ; il complète le niveau 2 avec la pose de verrous longs partagés en lecture.

Ainsi, le développeur peut contrôler la pose des verrous. Un choix autre que le degré 3 doit être effectué avec précaution dans les transactions de mise à jour, car il implique des risques d'incohérence. Seul en effet le degré 3 assure la sérialisabilité des transactions.

III.2.4.2 Le problème de l'interblocage : [8], [3]

Bien que la règle des deux phases garantisse la sérialisablilité des transactions, elle ne n'exclut pas la présence de blocages (*deadlocks*) ou d'exclusion d'un processus (*starvation*).

III.2.4.2.1 Définition

L'interblocage est aussi dit verrou mortel ; sa définition est la suivante :

Définition : **Interblocage** (*Deadlock*)

Situation dans laquelle un groupe de transactions est bloqué, chaque transaction du groupe attendant qu'une autre transaction du groupe relâche un verrou pour pouvoir continuer.

L'utilisation du verrouillage à deux phases n'est pas exempte du phénomène d'interblocage. A titre d'exemple voici un cas où cette situation peut se produire :

T1		T2	
	1. Lock(x,lecture)		3. Lock(y,lecture)
	2. lire(x)		4. lire(y)
T1 → bloquée	5. Lock(y,ecriture)	T2 → bloquée	6. Lock(x,ecriture)
	écrire(y)		écrire(x)
	…		…

Fig III.n : *situation d'interblocage entre T1 et T2* [3]

Dans l'exécution concurrente de la figure III.n, à l'instant 5 la transaction T1 est mise attente d'un verrou en mode écriture sur y : car un verrou en mode lecture (incompatible) est détenu par T2. A l'instant 6 c'est T2 qui est mise en attente d'un verrou d'écriture sur x. Les deux transactions se trouvent bloquées, chacune attendant la libération d'un verrou détenu par l'autre.

III.2.4.2.2 Conditions pour provoquer un interblocage : [28]

Quatre conditions doivent se réunir pour provoquer un interblocage :

1. *Condition d'exclusion mutuelle* : chaque ressource (granule) est soit attribuée à un seul processus (transaction), soit disponible.

2. *Condition de détention et d'attente* : les processus ayant déjà obtenu des ressources peuvent en demander de nouvelles.

3. *Pas de réquisition* : les ressources déjà détenues ne peuvent être retirées de force à un des processus. Elles doivent être explicitement libérées par le processus qui les détient.

4. *Condition d'attente circulaire* : il doit y avoir un circuit d'au moins deux processus, chacun attendant une ressource détenue par un autre processus du circuit.

III.2.4.2.3 Représentation de l'interblocage : [8]

Nous présentons ci-dessous deux types de graphes représentant les interblocages : le graphe des attentes et le graphe des allocations.

a) Graphe des attentes :

Le graphe des attentes est un graphe G (T,Γ) où T est l'ensemble des transactions concurrentes {T1, T2, … Tn} se partageant les granules G1, G2, … Gm et Γ est la relation « attend » définie par : Tp « attend » Tq si et seulement si Tp attend le verrouillage d'un objet Gi alors que cette requête ne peut être acceptée parce que Gi a déjà été verrouillé par Tq.

Définition : **Graphe des attentes** (*Waiting graph*)

> Graphe dont les nœuds correspondent aux transactions et les arcs représentent les attentes entre les transactions.

Le théorème suivant a été introduit dès 1968 [23] : il existe une situation de verrou mortel si et seulement si le graphe des attente possède un circuit. La figure III.o illustre ce théorème sur l'exemple introduit ci-dessus. La preuve est simple. En effet si le graphe des attentes possède un circuit, c'est qu'il existe un groupe de transactions tel que T1 attend T2, T2 attend T3, … , Tk attend T1. Chaque transaction du groupe est donc bloquée en attente d'un objet du fait de l'utilisation de cet objet par une autre transaction du groupe. La fin de l'exécution de toutes les transactions n'appartenant pas au groupe ne permet donc pas de débloquer une transaction du groupe.

Fig III.o : *Exemple de graphe d'attente avec circuit*

Réciproquement, l'existence d'une situation de verrou mortel implique l'existence d'au moins un circuit. S'il n'en était pas ainsi, tout groupe de transactions serait tel que le sous-graphe des attentes qu'il engendre ne possèderait pas de circuit. Après exécution de toutes les transactions n'appartenant pas au groupe, il serait donc possible de débloquer une transaction du groupe puisqu'un graphe sans circuit possède au moins un sommet pendant. Toutes les transactions appartenant à un circuit sont en situation de verrou mortel ; de plus une transaction attendant une transaction en situation de verrou mortel est elle-même en situation de verrou mortel (voir figure III.p).

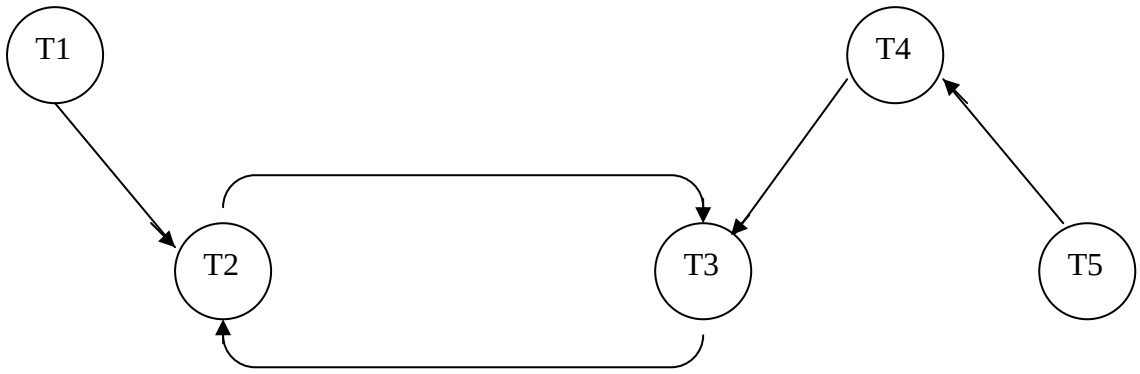


Fig III.p : *Transactions en situation de verrou mortel*

Il est intéressent d'établir le rapport entre graphe des attentes et graphe de précédence. Par définition si une transaction Ti `attend` une transaction Tj, alors Tj a verrouillé un objet O dont le verrouillage est demandé par Ti en mode incompatible. Ainsi l'opération pour laquelle Tj a verrouillé O sera exécutée avant celle demandée par Tj car les deux opérations sont incompatibles et donc non permutables. Donc Tj `précède` Ti. Toutefois, la relation de précédence n'implique généralement pas la relation d'attente. Donc en changeant l'orientation des arcs du graphe des attentes, on obtient un sous-graphe du graphe de précédence. Cela implique que si le graphe des attentes a un circuit, il en sera de même pour le graphe de précédence. En conséquence, une situation de verrou mortel ne peut pas donner lieu à une exécution sérialisable même s'il était possible de terminer les transactions interbloquées.

b) Graphe des allocations :

Le graphe des allocations est composé de deux ensembles de sommets :

1. l'ensemble T des transactions
2. l'ensemble O des objets.

Un arc relie l'objet Oi à la transaction Tp si et seulement si Tp a obtenu le verrouillage de Oi dans au moins un mode d'opération ; l'arc est valué par les modes d'opérations alloués. Un arc relie la transaction Tp à l'objet Oi si et seulement si Tp a demandé et n'a pas encore obtenu l'allocation de ce granule ; l'arrc est valué par les modes d'opérations demandés. La figure III.r représente le graphe des allocations de l'exemple de la figure III.q.

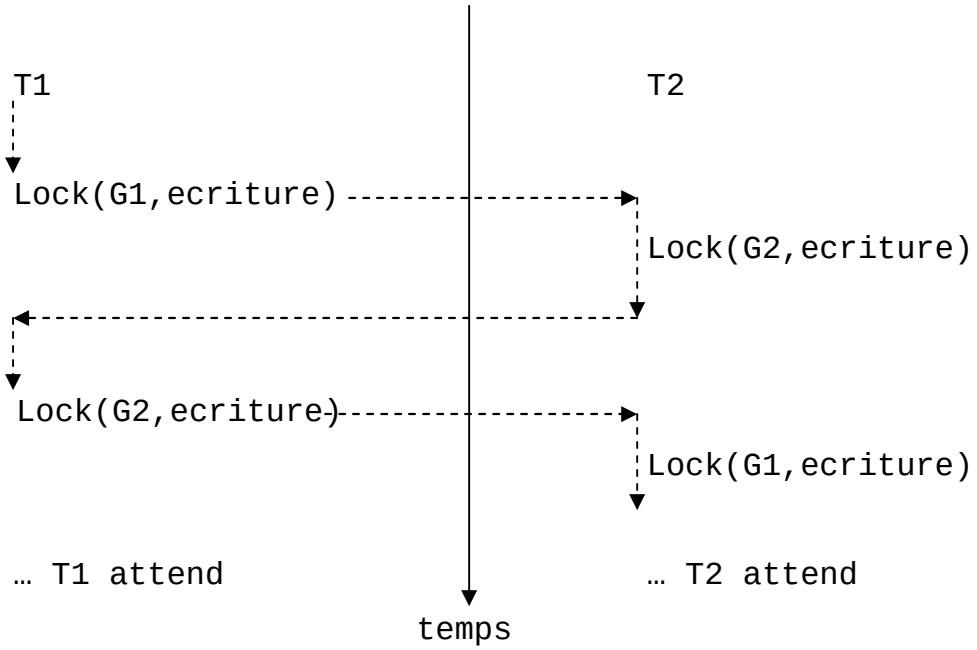


Fig III.q : *exemple d'interblocage entre T1 et T2*

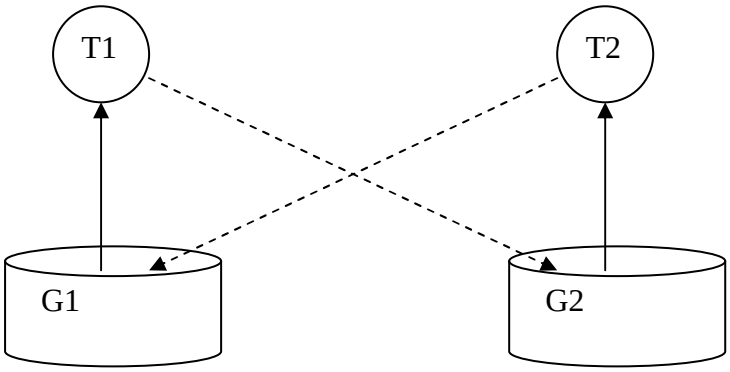


Fig III.r : *exemple de graphe des allocations*

Il est simple de démontrer le théorème suivant : une condition nécessaire d'existence de verrou mortel est la présence d'un circuit sur le graphe des allocations. Cette condition n'est en général pas suffisante. La preuve s'effectue par l'absurde. En effet, il est possible de prouver que s'il n'existe pas de circuit sur le graphe des allocations, il ne peut y avoir d'interblocage. En effet, soit T un groupe quelconque de transactions. Du fait que le graphe des allocations est sans circuit, le sous-graphe obtenu après exécution des transactions n'appartenant pas à T est sans circuit. Il possède donc un sommet pendant. Ce sommet ne peut être un granule car un granule non verrouillé ne peut être attendu. Dans tout groupe de transactions T, l'exécution supposée de toutes les transactions n'appartenant pas au groupe T conduit donc à debloquer une transaction du groupe. Il n'y a donc pas situation de verrou mortel.

Il faut remarquer que la condition est suffisante dans le cas où les seuls modes de lecture et d'écriture sont distingués. Ceci permet en général de détecter les situations de verrou mortel par détection de circuit dans le graphe des allocations dans la plupart des systèmes classiques. Nous pouvons noter qu'en général il n'y a pas de rapport direct entre le graphe de précédence et le graphe des allocations. Cependant, si les seuls modes existants sont lecture et écriture, la présence d'un circuit dans le graphe des allocations est équivalente à l'existence d'une situation de verrou mortel et donc à celle d'un circuit dans le graphe des attentes. Sous cette condition, la présence d'un circuit dans le graphe des allocations entraîne ainsi celle d'un circuit dans le graphe de précédence.

III.2.4.2.4 Prévention de l'interblocage : [28], [8]

Revenons aux quatre conditions mentionnées en III.2.4.2.2. Si nous pouvons garantir qu'au moins une de ces conditions ne sera jamais satisfaite, les interblocages seront impossibles.

a) S'attaquer à la condition de l'exclusion mutuelle :

Si une ressource (granule) n'est jamais attribuée de manière exclusive, nous n'aurons jamais d'interblocage. Toutes fois il est également clair que le fait de permettre à deux processus (transactions) d'écrire vers une imprimante au même moment peut conduire à un désastre, puisque le spoule d'impression permet à plusieurs processus de générer des résultats simultanément. Dans ce modèle le seul processus qui demande réellement l'imprimante physique est le démon d'impression. Comme celui-ci ne sollicite jamais aucune autre ressource, nous pouvons éliminer l'interblocage de l'imprimante.

Malheureusement, il n'est pas possible de traiter tous les périphériques en différé (la table des processus ne s'y prête pas). De plus la concurrence pour l'espace disque nécessaire au traitement en différé peut elle-même conduire à un interblocage. Que se passerait-il si deux processus remplissaient chacun la moitié de l'espace disponible pour le spoule sans avoir terminé ? Dans un cas où le démon commence à imprimer avant qu'un processus ne génère tous ses résultats, l'imprimante peut rester bloquée pendant des heures si ce processus interrompt la sortie des résultats. C'est pourquoi les démons sont normalement programmés pour imprimer uniquement après que l'ensemble du fichier de sortie est disponible. Dans ce cas, nous avons deux processus qui ont terminé partiellement, mais pas totalement, leur sortie, et qui ne peuvent pas continuer : aucun processus ne finira jamais. Nous avons donc un interblocage sur le disque.

Toutefois, on trouve là l'ébauche d'une idée souvent applicable. Eviter d'attribuer une ressource lorsque cela n'est pas absolument nécessaire et s'assurer que le nombre de processus susceptibles de réclamer la ressource est aussi faible que possible.

b) S’attaquer à la condition de détention et d’attente :

La deuxième des conditions mentionnées plus haut, dite condition de détention et d’attente (*hold and wait condition*) semble un peu plus prometteuse. Si nous pouvons empêcher que des processus qui détiennent des ressources attendent d’autres ressources, nous pouvons éviter les interblocages. Pour ce faire, il est possible d’exiger de tous les processus qu’ils demandent les ressources dont ils ont besoin avant de commencer à s’exécuter. Si elles sont toutes disponibles, le processus se verra attribuer tout ce dont il a besoin et pourra s’exécuter jusqu’au bout. Si une ou plusieurs ressources sont indisponibles, rien ne pourra être attribué et le processus se contentera d’attendre.

De cette approche découle un problème immédiat : bon nombre de processus ne savent pas de combien de ressources ils vont avoir besoin avant de commencer à s’exécuter. En fait, s’ils le savaient, il serait possible d’employer un algorithme connu : l’algorithme du banquier. Un autre problème réside dans le fait que les ressources ne seront pas utilisées de manière optimale. Prenons par exemple un processus qui lit les données sur une bande, les analyse pendant une heure puis écrit la bande de sortie et imprime les résultats sur une table traçante. Si toutes les ressources devaient être demandées par avance, le processus monopoliserait le dérouleur de bandes et la table traçante pendant une heure.

Néanmoins, certains mainframes qui fonctionnent en traitement par lots demandent à l’utilisateur de répertorier toutes les ressources au début de chaque tâche. Le système acquiert alors immédiatement toutes les ressources et les conserve jusqu’à la fin. Bien qu’une telle mise en œuvre soit lourde à programmer et constitue un gaspillage en termes de ressources, elle évite les interblocages.

Pour contrecarrer de manière légèrement différente la condition *hold-and-wait*, nous pouvons exiger d’un processus qui demande une ressource qu’il commence par libérer provisoirement toutes les ressources qu’il détient. Il peut ensuite tenter d’obtenir tout ce dont il a besoin en une seule fois.

c) S’attaquer à la condition de non-préemption :

S’attaquer à la condition de non-préemption est moins simple. Si un processus a été attribué à l’imprimante et qu’il se trouve à mi-chemin dans son travail d’impression, le fait de lui retirer l’imprimante de force parce que la table traçante demandée n’est pas disponible est, au mieux, très délicat, et au pire impossible.

d) S'attaquer à la condition de l'attente circulaire :

Nous voici arrivés à la dernière condition. Il existe plusieurs manières d'éliminer l'attente circulaire. On peut simplement établir une règle qui stipule qu'un processus est à tout moment attribué à une seule ressource. S'il a besoin d'une seconde ressource, il doit libérer la première. Pour un processus dont le travail consiste à recopier un fichier volumineux d'une bande vers l'imprimante, cette restriction est inacceptable.

Une autre manière d'éviter l'attente circulaire consiste à fournir un dénombrement global des ressources. Maintenant voici la règle : les processus peuvent demander des ressources dès qu'ils le souhaitent, mais toutes les demandes doivent être effectuées selon un ordre numérique. Un processus peut demander une table traçante, puis un dérouleur de bandes, mais il ne peut procéder dans l'ordre inverse.

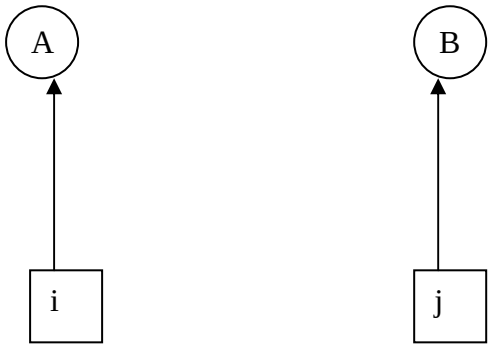


Figure III.s : *un graphe de ressources*

Avec cette règle, le graphe d'allocations de ressources ne peut jamais connaître de cycle. Voyons pourquoi cela est vrai dans la cas des deux processus de la figure III.s. Nous pouvons avoir un interblocage uniquement si A demande la ressource j et B la ressource i. Si i et j sont des ressources distinctes, elles auront des numéros différents. Si i > j , alors A n'est pas autorisé à demander j car son numéro est inférieur à celui de la ressource qu'il détient déjà. Si i < j , alors B n'est pas autorisé à demander i car son numéro est inférieur à celui de la ressource qu'il possède déjà. Dans tous les cas de figure, l'interblocage est impossible.

Cette logique est également valable avec plusieurs processus. A tout instant , une des ressources assignées a le numéro le plus élevé, or le processus qui détient cette ressource ne demandera jamais une ressource déjà assignée. Soit il se terminera, soit au pire, il demandera des ressources de numéro plus élevé, qui seront toutes disponibles. Finalement, il se terminera et libérera ses ressources. A ce stade, un autre processus détiendra la ressource la plus grande et pourra également se terminer. En résumé, il existe un scénario dans lequel tous les processus se terminent et où aucun interblocage ne se produit.

Une variation mineure de cet algorithme consiste à abandonner l'exigence selon laquelle les ressources sont acquises en une séquence strictement croissante et à insister simplement sur le fait qu'aucun processus ne demande une ressource inférieure à celle qu'il détient déjà. Si un processus demande au départ les ressources 9 et 10 puis les libère, il reprend effectivement tout au début. Il n'y a donc aucune raison pour lui interdire de demander la ressource 1.

Bien que le fait d'ordonnancer numériquement les ressources élimine le problème des interblocages, il peut s'avérer impossible de trouver un ordonnancement qui satisfasse tout un chacun. Lorsque les ressources incluent des positions dans la table des processus, de l'espace de spoule sur le disque, des enregistrements de base de données verrouillés et d'autres ressources abstraites, le nombre de ressources potentielles et d'utilisations différentes peut être si élevé qu'aucun ordonnancement ne puisse fonctionner.

e) estampillage des transactions :

Les paragraphes a), b), c) et d) précédent ont décrit l'ordonnancement des ressources. Il existe également l'ordonnancement des transactions qui est possible à partir d'une estampille.

Définition : **Estampille de transaction** (*Transaction Timestamp*)

> Numéro unique attribué à une transaction permettant de l'ordonner strictement par rapport aux autres transactions.

En général, l'estampille attribuée à une transaction est son horodate de lancement concaténée avec le numéro du processeur sur lequel elle est lancée, ceci afin d'empêcher l'égalité des estampilles pour deux transactions lancées au même instant : celles-ci diffèrent alors par le numéro du processeur en poids faibles. Le numéro du processeur n'est utile que dans les architectures parallèles.

A partir des estampilles deux algorithmes ont été proposés pour prévenir les verrous mortels. Tous deux consistent à défaire plus ou moins directement une transaction dans le cas d'attente, de sorte à ne permettre que des attentes sans risque de circuit. L'algorithme WAIT-DIE consiste à annuler les transactions qui demandent des ressources tenues par des transactions plus anciennes. La transaction la plus récente est reprise avec la même estampille ; elle finit ainsi par devenir ancienne et à passer. Il ne peut y avoir de verrou mortel, les seules attentes possibles étant dans l'ordre où une transaction ancienne attend une transaction récente. Le contrôle des attentes imposé par l'algorithme est précisé en figure III.t.

```
/* algorithme WAIT-DIE */
Procédure Attendre (Ti,Tj)
Début
   /* Ti réclame un verrou tenu par Tj */
   Si ts(Ti) < ts(Tj) alors Ti waits sinon Ti dies finsi
Fin
```

Figure III.t : *Contrôle des attentes dans l'algorithme WAIT-DIE* [8]

L'algorithme WOUND-WAIT est un peu plus subtil. Tout type d'attente est permis. Mais si une transaction plus ancienne attend une plus récente, la récente est blessée (*wounded*), ce qui signifie qu'elle ne peut plus attendre : si elle réclame un verrou tenu par une autre transaction elle automatiquement défaite et reprise. Le contrôle des attentes imposé par l'algorithme est représenté en figure III.u ; une transaction blessée ne peut donc attendre.

```
/* algorithme WOUND-WAIT */
Procédure Attendre (Ti,Tj)
Début
   /* Ti réclame un verrou tenu par Tj */
   Si ts(Ti) < ts(Tj) alors Tj is wounded sinon Ti waits finsi
Fin
```

Figure III.u : *Contrôle des attentes dans l'algorithme WOUND-WAIT* [8]

III.2.4.2.5 Détection de l'interblocage : [8]

La prévention provoque en général trop de reprises de transactions, car les méthodes défont des transactions alors que des verrous mortels ne sont pas sûrs d'apparaître. Au contraire, la détection laisse le problème se produire, détecte les circuits d'attente et annule certaines transactions afin de rompre les circuits d'attente.

Un algorithme de détection de l'interblocage peut se déduire d'un algorithme de détection de circuits appliqué au graphe des attentes ou des allocations. Nous présentons ici une mise en œuvre de l'algorithme qui consiste à tester si un graphe est sans circuit par élimination successive des sommets pendants.

Sur le graphe des attentes, un sommet est pendant si la transaction qu'il représente n'attend le verrouillage d'aucun granule. Soit N(k) le nombre de granules dont la transaction Tk attend le verrouillage. Une première réduction du graphe peut être obtenue par élimination des sommets pendants, donc tels que N(k)=0.

Le problème est alors de recalculer les nombres de granules attendus N(k) après réduction pour pouvoir effectuer la réduction suivante. Ceci peut être fait en comptant les demandes qui peuvent être satisfaites après chaque réduction, et n décrémentant N(k) chaque fois que l'on compte une demande de la transaction Tk. L'application de la méthode nécessite deux précautions :

1. marquer les demandes comptées pour ne pas les compter deux fois ;
2. disposer d'une procédure permettant de tester si une demande peut être satisfaite compte tenu de l'état des verrouillages des transactions non encore éliminées du graphe des attentes.

Soit donc `SLOCK(k,G,M)` une procédure booléenne permettant de tester si la demande du granule G en mode M de la transaction Tk peut être satisfaite compte tenu de l'état d'allocation des granules aux transactions présentes dans le graphe des attentes. Cette procédure répond `VRAI` si la demande peut être satisfaite et `FAUX` sinon. Le code de cette procédure est analogue à celui de l'algorithme `LOCK` vu ci-dessus, à ceci près que seules les transactions de T sont prises en compte (les autres sont supposées exécutées et terminées) et que l'état de verrouillage n'est pas modifié. La figure III.v présente une procédure `DETECTER` répondant `VRAI` s'il y a situation de verrou mortel et `FAUX` sinon. Cette procédure élimine donc progressivement les transactions pendantes du graphe des attentes.

```
Procédure Detecter : booléen
Début
  T = { Liste des transactions telles que N(k)≠0 }
  G = { Liste des granules alloués aux transactions dans T }
  Pour chaque entrée g de G faire
    Pour chaque demande non marquée M de Tk en attente de g
    faire
      Si SLOCK(k,g,Q) = VRAI alors
        Marquer Q
        N(k) := N(k) - 1
        Si N(k) = 0 alors
          Eliminer Tk de T
          Ajouter les granules verrouillés par Tk à G
        Finsi
      Finsi
    FinPour
  Si T = Ø alors DETECTER := FAUX sinon DETECTER := VRAI
  finsi
Fin
```

Figure III.v : *Algorithme de détection de verrou mortel* [8]

Quand une situation d'interblocage est détectée, le problème qui se pose est de choisir une transaction à recycler de façon à briser les circuits du graphe des attentes. L'algorithme de détection présenté ci-dessus fournit la liste des transactions en situation d'interblocage. Il faut donc choisir une transaction de cette liste. Cependant, tous les choix ne sont pas judicieux, comme le montre la figure III.w. Une solution à ce problème peut être de recycler la transaction qui bloque le plus grand nombre d'autres transactions, c'est-à-dire qui correspond au sommet de demi-degré intérieur le plus élevé sur le graphe des attentes. Le choix de la transaction à reprendre doit aussi chercher à minimiser le coût de reprise.

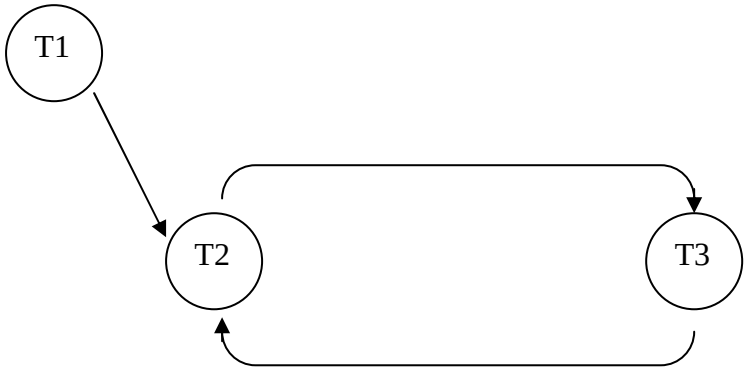


Figure III.w : *Exemple de choix difficile de transaction à reprendre*

Le coût d'une solution de type détection avec reprise peut être réduit. En effet, il est possible de déclencher un algorithme détection seulement quand une transaction attend un verrouillage depuis un temps important (par exemple quelques secondes), plutôt qu'à chaque début d'attente.

D'autres algorithmes de détection sont possibles. Le graphe d'allocation est souvent utilisé dans les systèmes répartis. Lors d'une attente qui dure, un algorithme envoie une enquête le long des arcs du graphe des allocations. Cette enquête est transmise au granule attendu, puis aux transactions bloquant ce granule, puis aux granules attendus s'il en existe, etc. Si l'enquête revient à la transaction initiale, c'est qu'il y a un verrou mortel. Cet algorithme est moins efficace en centralisé.

III.2.4.2.6 L'évitement de l'interblocage : [28]

Dans la plupart des systèmes les ressources sont demandées l'une après l'autre. Le système doit déterminer si l'attribution de la ressource est sûre t ne l'accorder que dans ce cas. On peut ainsi se demander s'il existe un algorithme susceptible d'éviter systématiquement les interblocages en faisant toujours le bon choix. La réponse tient en un « oui » mitigé. Il est possible d'éviter les interblocages, mais uniquement si nous possédons préalablement certaines informations. Dans cette section nous allons étudier les méthodes qui permettent d'éviter les interblocages grâce à une attribution réfléchie des ressources.

a) Les trajectoires de ressources :

Les principaux algorithmes destinés à éviter les interblocages reposent sur le concept des états sûrs (*safe states*). Avant de les décrire nous allons présenter le concept de sûreté au moyen d'un graphique simple à comprendre. Bien que cette approche ne se traduise pas directement en un algorithme utilisable, elle donnera un aperçu direct de la nature du problème.

A la figure III.x, nous voyons un modèle destiné à gérer deux processus et deux ressources, comme une imprimante et une table traçante. L'axe horizontal représente le nombre d'instructions exécutées par le processus A. L'axe vertical représente le nombre d'instructions exécutées par le processus B. A I_1, A demande une imprimante et à I_2, il a besoin d'une table traçante. L'imprimante et la table traçante sont libérées respectivement à I_3 et I_4. Le processus B a besoin de la table traçante de I_5 à I_7 et l'imprimante de I_6 à I_8.

Chaque point du diagramme représente un état commun aux deux processus. Au départ, l'état se trouve à p ; aucun processus n'exécute une instruction. Si l'ordonnanceur de tâches décide d'exécuter A en premier, nous arrivons au point q, où A a exécuté un certain nombre d'instructions, mais où B n'en a réalisé aucune. Au point q la trajectoire devient verticale, ce qui indique que l'ordonnanceur a choisi d'exécuter B. Avec un seul processeur, tous les chemins doivent être horizontaux ou verticaux, jamais diagonaux. En outre, le mouvement se fait toujours dans le sens nord ouest, jamais sud ouest (les processus ne peuvent s'exécuter à l'envers).

Lorsque A franchit la ligne I1 sur le chemin allant de r à s, il demande et obtient l'imprimante. Lorsque B atteint le point t, il demande la table traçante.

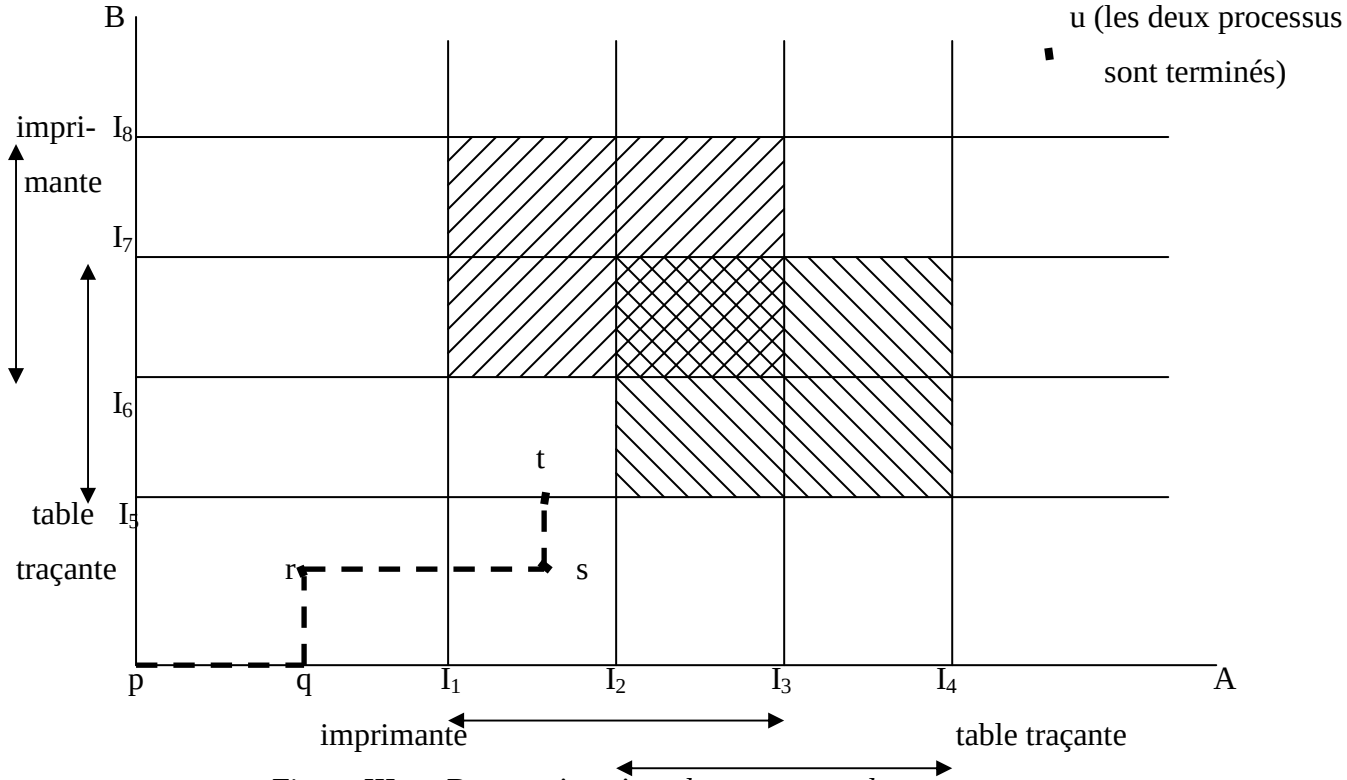


Figure III.x : *Deux trajectoires de ressources de processus*

Les zones hachurées sont particulièrement intéressantes. Celle qui est hachurée dans le sens sud-ouest/ nord-est représente les deux processus qui détiennent l'imprimante. La règle de l'exclusion mutuelle fait qu'il est impossible de pénétrer dans cette zone. De même, la région hachurée dans l'autre sens représente les deux processus qui détiennent la table traçante. Elle est également inaccessible.

Si le système pénètre dans la zone délimitée de part et d'autre par I_1 et I_2 et par I_5 et I_6 en haut et en bas, il se bloquera finalement à l'intersection de I_2 t I_6. A ce stade A demande la table traçante et B l'imprimante. Ces deux ressources sont attribuées. L'ensemble de la zone n'est pas sûr et on ne doit pas y pénétrer. Au point t, la seule opération sûre est l'exécution du processus A jusqu'à I_4. Au-delà, toute trajectoire vers u est sûre.

Ici, le point important à noter est que, au point t, B demande une ressource. Le système doit choisir de l'attribuer ou non. S'il le fait, il pénétrera dans une région non sûre et aboutira finalement à un interblocage. Pour l'éviter, il est nécessaire de suspendre B jusqu'à ce que A ait demandé et libéré la table traçante.

b) Les états sûrs et non sûrs :

Les algorithmes d'évitement des interblocages que nous étudierons utilisent les informations sur les allocations en cours et demandes des différents processus/ressources :

Ressources existantes

$(E_1, E_2, \ldots, E_m)$

Ressources disponibles

$(A_1, A_2, \ldots, A_m)$

Matrice des allocations en cours

$$\begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$$

Matrice de demandes

$$\begin{bmatrix} R_{11} & R_{12} & \ldots & R_{1m} \\ R_{21} & R_{22} & \ldots & R_{2m} \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ R_{n1} & R_{n2} & \ldots & R_{nm} \end{bmatrix}$$

A tout instant, il existe un état en cours composé de E, A, C et R. On dit de l'état A qu'il est sûre (*safe*) s'il n'est pas bloqué et qu'il existe un ordonnancement selon lequel chaque processus peut s'exécuter jusqu'au bout, même si tous demandent d'un seul coup leur nombre maximum de ressources.

Dans la figure III.y(a), nous sommes en présence d'un état où A possède 3 instances de la ressource mais peut au final avoir besoin de 9. B en possède actuellement 2 et pourra avoir besoin ultérieurement de 4 simultanément. De même C en possède 2 mais pourra avoir besoin de 5 instances de ressources supplémentaires. Il existe au total 10 instances de ressources : ainsi avec 7 ressources déjà attribuées, 3 sont toujours libres.

	A	Max
A	3	9
B	2	4
C	2	7

Libres : 3

(a)

	A	Max
A	3	9
B	4	4
C	2	7

Libres : 1

(b)

	A	Max
A	3	9
B	0	-
C	2	7

Libres : 5

(c)

	A	Max
A	3	9
B	0	-
C	7	7

Libres : 0

(d)

	A	Max
A	3	9
B	0	-
C	0	-

Libres : 7

(e)

Figure III.y : *Démonstration que l'état de (a) est sûr*

L'état de la figure III.y(a) est sûr car il existe une séquence d'allocations qui permet à tous les processus de s'exécuter jusqu'au bout. Ainsi l'ordonnanceur peut se contenter d'exécuter uniquement B jusqu'à ce qu'il demande et obtienne deux instances de ressources supplémentaires, ce qui conduit à la figure III.y (b). Lorsque B s'achève, nous obtenons l'état de la figure III.y(c). L'ordonnanceur peut alors exécuter C, ce qui mène finalement à la figure III.y(d). Lorsque C s'achève, nous obtenons la figure III.y(e). A présent A peut obtenir les 6 instances de ressources dont il a besoin et s'exécuter jusqu'au bout. Ainsi l'état de la figure III.y(a) est sûr car, grâce à un ordonnancement attentif, le système peut éviter les interblocages.

Supposons maintenant qu'avec l'état initial illustré à la figure III.z(a), A demande et obtienne une autre ressource, nous donnant la figure III.z(b). Pouvons nous trouver une séquence qui soit sûre de fonctionner ? Essayons. L'ordonnanceur peut exécuter B jusqu'à ce qu'il demande toutes ses ressources comme le montre la figure III.z(c).

	A	Max
A	3	9
B	2	4
C	2	7

Libres : 3

(a)

	A	Max
A	4	9
B	2	4
C	2	7

Libres : 2

(b)

	A	Max
A	4	9
B	4	4
C	2	7

Libres : 0

(c)

	A	Max
A	4	9
B	-	-
C	2	7

Libres : 4

(d)

Figure III.z : *Démonstration que l'état de (b) n' est pas sûr*

Finalement B s'achève et nous obtenons la situation illustrée à la figure III.z(d). A ce stade, nous sommes bloqués. Il ne nous reste que 4 instances de ressources libres et chacun des processus actifs en demande 5.

Aucune séquence n'est assurée d'arriver à son terme. Ainsi, le mode d'allocation des ressources que nous avons choisi et qui nous a conduit du système de la figure III.z(a) à celui de la figure III.z(b) nous a fait passer d'un état sûr à un état non sûr. Exécuter ensuite A ou C en commençant à la figure III.z(b) ne fonctionne pas non plus. Rétrospectivement, la demande de A n'aurait pas dû être satisfaite.

Il convient de noter qu'un état non sûr ne conduit pas nécessairement à un interblocage. Ainsi, à partir de la figure III.z(b), le système peut s'exécuter pendant un certain temps. En fait, un processus peut même s'achever. De plus, il est possible que A libère une ressource avant d'en demander d'autres, ce qui permet à C de se terminer et d'éviter par là même un interblocage. Ainsi la différence entre un état sûr et un état non sûr tient à ce que dans un état sûr, le système peut garantir que tous les processus s'achèveront ; dans un état non sûr, cette garantie ne peut pas être accordée.

c) L'algorithme du banquier pour une ressource unique :

En 1965, Dijkstra a proposé un algorithme d'ordonnancement qui permet d'éviter les interblocages, appelé **algorithme du banquier**. Il s'inspire de la manière dont un banquier d'une petite ville accorde des crédits à un groupe de clients. L'algorithme vérifie si le fait d'accorder cette requête conduit à un état non sûr. Si tel est le cas la requête est refusée. Dans le cas contraire elle est accordée. Dans la figure III.aa(a) quatre clients A, B, C et D ont reçu chacun un crédit d'un certain montant n unités (par exemple 1 unité égale 1000 €). Le banquier sait que les clients n'auront pas tous besoin immédiatement de leur crédit maximum. Il leur a donc réservé seulement 10 unités au lieu de 22 (dans cette analogie, les clients sont les processus, les unités sont par exemple les dérouleurs de bandes et le banquier est le système d'exploitation).

	A	Max
A	0	6
B	0	5
C	0	4
D	0	7

Libres : 10

(a)

	A	Max
A	1	6
B	1	5
C	2	4
D	4	7

Libres : 2

(b)

	A	Max
A	1	6
B	2	5
C	2	4
D	4	7

Libres : 1

(c)

Figure III.aa : *trois états d'allocation de ressources : (a) sûr, (b) sûr, (c) non sûr*

Les clients vaquent donc à leur affaires en réalisant de temps à autre des emprunts (c'est-à-dire en demandant des ressources). A un moment donné, la situation peut être celle de la figure III.aa(b). Cet état set sûr car, avec 2 unités non allouées le banquier peut mettre en attente toutes les requêtes, à l'exception de celle de C. C peut alors se terminer et libérer les 4 ressources qu'il détient. Fort de unités, le banquier peut ensuite satisfaire D ou B, et ainsi de suite.

Voyons ce qui se passe si B réclame une unité de plus et l'obtient (voir figure III.a(b)). Cette situation, illustrée à la figure III.aa(c), n'est pas sûre. Si tous les clients demandent en même temps leur crédit maximum, le banquier ne peut satisfaire aucun de ses clients. Un état non sûr ne conduit pas nécessairement à un interblocage, car un client peut ne pas demander son crédit maximum, mais le banquier ne doit pas compter sur cette éventualité.

L'algorithme du banquier consiste à examiner chaque nouvelle requête pour voir si elle conduit à un état sûr. Le cas échéant, la ressource est allouée, sinon, elle est mise en attente. Afin de voir si un état est sûr, le banquier vérifie s'il possède suffisamment de ressources pour satisfaire un client. Si tel est le cas, on suppose que ses crédits seront remboursés, on examine le cas du client le plus proche de la limite, et ainsi de suite. Si tous les crédits sont finalement remboursés, l'état est sûr et la requête initiale peut être accordée.

d) L'algorithme du banquier pour plusieurs ressources :

Il est possible de généraliser l'algorithme du banquier à plusieurs ressources. La figure III.ab illustre son fonctionnement.

Pr	R1	R2	R3	R4
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Ressources attribuées

Pr	R1	R2	R3	R4
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Ressources toujours en attente

Pr : Processus ; R1 : dérouleurs de bandes ; R2 : tables traçantes ; R3 : Scanners ; R4 : CD-ROM

E=(6342) P=(5322) A=(1020)

Figure III.ab : *algorithme du banquier pour plusieurs ressources*

La figure III.ab nous donne deux matrices. Celle de gauche illustre le nombre de ressources attribuées à chacun des processus et celle de droite montre le nombre de ressources dont chaque processus a besoin pour se terminer.

Ces matrices correspondent aux ressources C et R du paragraphe b) précèdent. Comme dans le cas d'une ressource unique, les processus doivent indiquer leur besoin total en ressources avant de s'exécuter, de sorte que le système puisse calculer à chaque instant la matrice de droite.

Parmi les trois vecteurs E,P et V ; E correspond aux ressources existantes, P aux ressources détenues et A aux ressources disponibles. Dans E, nous voyons que le système possède six dérouleurs de bandes, trois tables traçantes, quatre imprimantes et deux lecteurs de CD-ROM. Parmi ces ressources, cinq dérouleurs de bandes, trois tables traçantes, deux imprimantes et deux lecteurs de CD-ROM sont actuellement assignés. Ce résultat apparaît lorsqu'on ajoute les quatre colonnes de ressources de la matrice de gauche. Le vecteur de ressources disponibles correspond simplement à la différence entre les ressources détenues par le système et celles en cours d'utilisation. L'algorithme qui vérifie la sûreté de l'état peut maintenant être établi.

1. Recherchez une rangée dont les demandes de ressources non satisfaites sont inférieures ou égales à A. Si cette rangée n'existe pas, le système peut finalement se bloquer car aucun processus ne peut s'exécuter jusqu'au bout.
2. Supposons que le processus de la rangée choisie demande toutes les ressources dont il a besoin (nous avons la garantie que cela est possible) et qu'il se termine. Marquez ce processus comme achevé et ajoutez toutes ses ressources au vecteur A.
3. Recommencez les étapes 1 et 2, soit jusqu'à ce que tous les processus soient marqués comme terminés – auquel cas l'état initial est sûr –, soit jusqu'à ce qu'un interblocage se produise – auquel cas l'état initial n'était pas sûr.

S'il existe plusieurs processus éligibles lors de l'étape 1, peu importe celui qu l'on choisit : le pool des ressources disponibles s'accroît ou, au pire, reste identique.

Revenons à présent à l'exemple de la figure III.ab, où l'état en cours est sûr. Supposons maintenant que le processus B demande une imprimante. Cette requête peut être satisfaite car l'état qui en résulte est toujours sûr (le processus D peut se terminer, suivi du processus A ou E, puis des autres).

Après avoir attribué à B l'une des deux imprimantes restantes, imaginons alors que E sollicite la dernière imprimante. Le fait de satisfaire cette requête réduit le vecteur des ressources disponibles à (1 0 0 0), ce qui conduit à un interblocage. Il apparaît clairement que la requête de E doit être reportée pour un temps.

Remarque : L'algorithme est pour l'essentiel inutilisable car les processus savent rarement par avance quels vont être leurs besoins maximaux en termes de ressources. En outre, le nombre de processus n'est pas fixe, mais varie dynamiquement lorsque de nouveaux utilisateurs se connectent ou se déconnectent. De plus, les ressources que l'on croyait disponibles peuvent soudain disparaître (le dérouleur de bandes peut tomber en panne). Ainsi, dans la pratique, peu de systèmes existants, pour ne pas dire aucun, utilisent l'algorithme du banquier pour éviter les interblocages.

III.2.4.3 Autres problèmes soulevés par le verrouillage : [8]

Un autre problème soulevé par le verrouillage est le problème de **famine**, encore appelé **blocage permanent**. Ce problème survient dès qu'un groupe de transactions se coalise, en effectuant des opérations compatibles entre elles (par exemple des lectures), contre un transaction individuelle qui désire effectuer une opération incompatible avec les précédentes (par exemple une écriture). La transaction individuelle peut alors attendre indéfiniment. Les solutions à ce problème consistent en général à mettre en file d'attente les demandes de verrouillage dans leur ordre d'arrivée et n'accepter une requête de verrouillage que si elle est compatible avec les verrouillages en cours et ceux demandés par les requêtes les plus prioritaires en attente. Il faut noter que les algorithmes de prévention DIE-WAIT et WOUND-WAIT ne conduisent jamais une transaction à l'attente infinie. En effet, une transaction qui meurt garde son ancienne estampille lorsqu'elle est relancée. Elle devient ainsi plus vieille et finit toujours par passer, le principe étant d'avorter les transactions les plus jeunes.

Le problème des **fantômes** a également été soulevé []. Il survient lorsqu'un objet est introduit dans la base de donnée et ne peut être pris en compte par une transaction en cours qui devrait logiquement le traiter. Par exemple, soit la base de données de réservation de places d'avions, représentée en figure III.ac, composée de deux relations : PASSAGER(nom, numéro de vol, numéro de siège) et OCCUPATION (numéro de vol, nombre de passagers). Considérons maintenant les transactions suivantes :

- T1 (1ere partie) : lister la relation PASSAGER en lisant tuple à tuple ;
- T1 (2eme partie) : lister la relation OCCUPATION d'un seul tuple ;
- T2 : insérer dans PASSAGER le tuple (Fantomas, 100, 13) et incrémenter le nombre de passagers du vol numéro 100.

Passagers	Nom	N° vol	N° siège
	Dubois	100	3
	Durand	100	5
	Dupont	100	10
	Martin	100	15

Occupation	N° vol	NbrePass
	100	4

Figure III.ac : *illustration du problème du fantôme*

Les transactions sont supposées verrouiller les tuples. Supposons que les transactions s'enchevêtrent dans l'ordre T1 (1ere partie), T2, T1 (2eme partie). C'est une exécution valide puisque T2 accède à un granule non verrouillé qui n'existe même pas lorsque T1 exécute sa 1ere partie : le tuple « Fantomas ». Toutefois le résultat de T1 est une liste de 4 noms alors que le nombre de passagers est 5. « Fantomas » est ici un fantôme pour T1.

Ce problème, ainsi que la difficulté de citer les granules à verrouiller, peuvent être résolus par la définition de granules logiques (dans l'exemple, les passagers du vol 100) au moyen des prédicats []. Le verrouillage par prédicat permet également de définir des granules de taille variable, ajustées aux besoins des transactions. Malheureusement, il nécessite des algorithmes pour déterminer si deux prédicats sont disjoints et ce problème de logique n'a pas de solution suffisamment efficace pour être appliqué dynamiquement lors du verrouillage des objets. De plus, les prédicats sont définis sur des domaines dont les extensions ne sont pas consultables dans la base pour des raisons évidentes de performance. Donc, il est très difficile de déterminer si deux prédicats sont disjoints ; par exemple, PROFESSION='ingénieur' et SALAIRE<7000 seront déterminés logiquement non disjoints, alors qu'ils le sont dans la plupart des bases de données. Le verrouillage par prédicat est donc en pratique source d'attentes inutiles et finalement inapplicable.

III.2.4.4 Les améliorations du verrouillage :

Malgré le grand nombre de solutions proposées par les chercheurs, les systèmes continuent à appliquer le verrouillage deux phases avec prévention et détection des verrous mortels. Les degrés d'isolation choisis par les transactions permettent de maximiser le partage des données en limitant le contrôle. Le verrouillage est cependant très limitatif. Un premier problème qui se pose est le choix de la granularité des objets à verrouiller. Au-delà, la recherche sur l'amélioration continue et des solutions parfois applicables ont été proposées.

III.2.4.4.1 Verrouillage à granularité variable : [8]

Une granularité variable est possible. La technique consiste à définir un graphe acyclique d'objets emboîtés et à verrouiller à partir de la racine dans un mode d'intention jusqu'aux feuilles désirées qui sont verrouillées en mode explicite. Par exemple, une transaction désirant verrouiller un tuple en mode écriture verrouillera la table en intension d'écriture, puis la page en intension d'écriture, et enfin le tuple en mode écriture. Les modes d'intension obéissent aux mêmes règles de compatibilités que les modes explicites, mais non compatibles entre eux. Le verrouillage en intention permet simplement d'éviter les conflits avec les modes explicites. Sur un même objet, les modes explicites règlent les conflits. La figure III.ad donne la matrice de compatibilité entre les modes lecture (L), écriture (E), intention de lecture (IL) et intention d'écriture (IE).

	L	E	IL	IE
L	V	F	V	F
E	F	F	F	F
IL	V	F	V	V
IE	F	F	V	V

Figure III.ad : *Compatibilités entre les modes normaux et d'intention*

Une telle méthode peut être appliquée dans les bases relationnelles, mais aussi objet. Le graphe d'inclusion pour une base de données objet peut être base, extension de classe, page ou groupe (cluster) et objet. Il est représenté en figure III.ae.

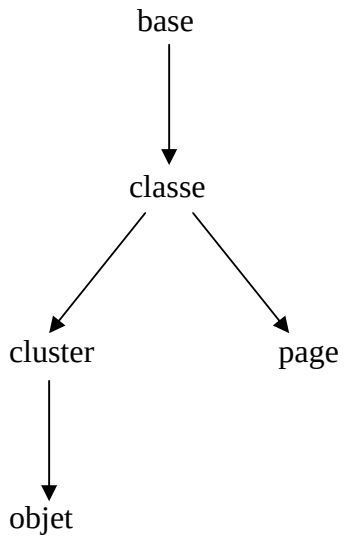


Figure III.ae : *granules variables pour une BD objet*

III.2.4.4.2 Verrouillage multi-versions : [8]

Le verrouillage multi-versions suppose l'existence d'au moins une version précédente d'un objet en cours de modification. C'est généralement le cas dans les systèmes puisque un journal des images avant mise à jour est géré en mémoire. Le principe est simple : lors d'un verrouillage en lecture, si le granule est occupé par une transaction en mode incompatible (donc en écriture en pratique), la version précédente du granule est délivrée à l'utilisateur. Une telle technique est viable lorsque les granules verrouillés sont des pages ou des tuples. Au-delà, il est difficile de constituer une version cohérente du granule rapidement.

Avec le verrouillage multi-versions, tout se passe comme si la transaction qui lit avait été lancée avant la transaction qui écrit. Malheureusement, la serialisabilité n'est pas garantie si la transaction qui accède à la version ancienne écrit par ailleurs. En effet, la mise à jour de la transaction en quelque sorte sautée n'est prise en compte par la transaction lisant, qui risque d'obtenir des résultats dépassés ne pouvant servir aux mises à jour. Seules les transactions n'effectuant que les lectures peuvent utiliser ce mécanisme, aussi appelé lecture dans le passé. Si l'on veut de plus garantir la reproductibilité des lectures, il faut gérer au niveau du système un cache des lectures effectuées par le passé, afin de les retrouver lors de la deuxième lecture.

III.2.4.4.3 Verrouillage altruiste : [8]

Le verrouillage altruiste suppose connu les pattern d'accès des transactions, c'est-à-dire au moins l'ordre d'accès aux granules et les granules non accédés. Il devient alors possible de relâcher les verrous tenus par une transaction longue lorsqu'on sait qu'elle n'utilisera plus un granule. Ce granule est alors ajouté à l'ensemble des granules utilisés par la transaction, appelée la **traînée** (*wake*) de la transaction. En cas de reprise, toutes les transactions dans la traînée d'une transaction sont aussi reprises (c'est-à-dire celles ayant accédé à des objets dans la traînée). C'est l'**effet domino**, selon lequel une transaction implique la reprise d'autres pour compenser la non isolation (voir ci-dessous). Par exemple, figure III.af, on voit que T2 peut s'exécuter alors que T1 n'est pas terminée car on sait que T1 ne reviendera pas sur c et que T2 n'accédera pas a. Si T3 est reprise, T4 doit l'être aussi car elle modifie c, lui-même modifié par T3.

Le verrouillage altruiste est difficile à appliquer en pratique car on ne connaît pas les patterns d'accès des transactions. De plus, l'effet domino reste mal maîtrisé. Cette technique pourrait être intéressante pour faire cohabiter des transactions longues avec des transactions courtes.

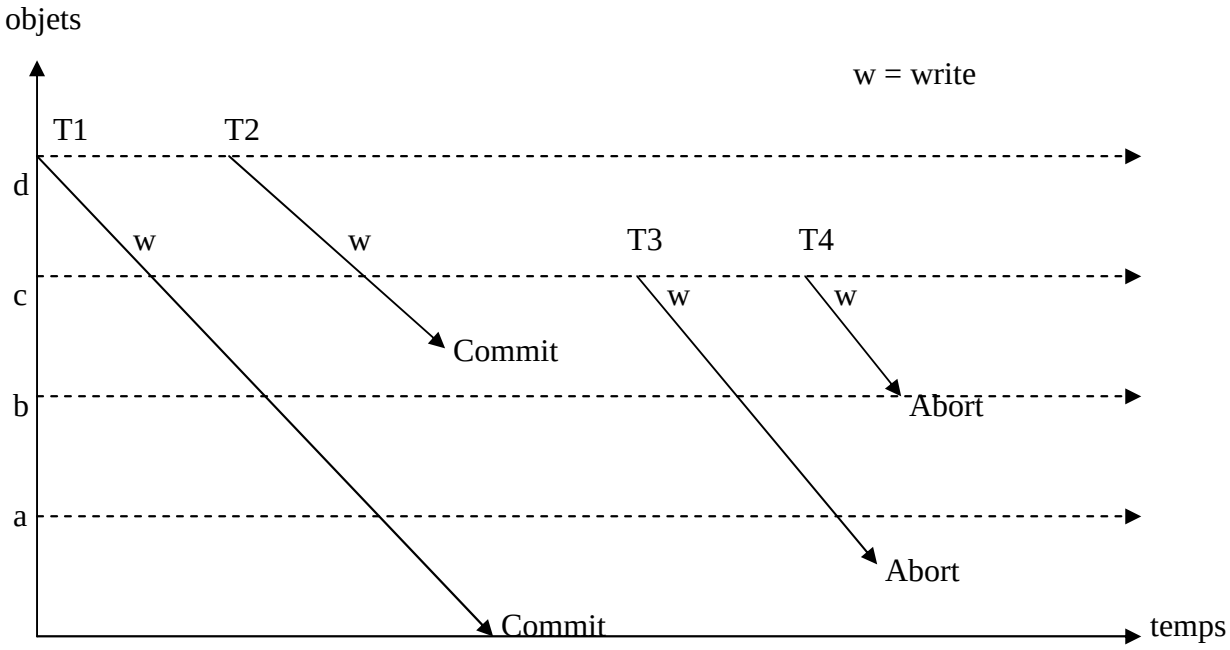


Figure III.af : *Exemple de verrouillage altruiste*

III.2.4.4.4 Commutativité sémantique d'opérations : [8]

Il est possible d'exploiter la **sémantique des opérations**, notamment dans les systèmes objet ou objet-relationnel où il existe des types (ou classes). Chaque type est caractérisé par une liste d'opérations (méthodes). Comme nous l'avons vu ci-dessus, les opérations commutatives sont permutables et n'entraînent pas de conflits de précédence. Il est donc intéressent de distinguer des modes de verrouillage plus fin que lecture et écriture, permettant de prendre en compte les opérations effectives sur les objets typés, et non pas les actions de base lecture et écriture.

Introduire la commutativité entre opérations est utile si les opérations de mises à jour à priori incompatibles, sont souvent commutatives. Si l'on regarde seulement le nom de l'opération, la commutativité est rare car elle dépend souvent des paramètres, notamment de la réponse. Les chercheurs ont donc introduit des modes d'opérations incluant les réponses. Par exemple, avec des ensembles, il est intéressent de distinguer insérer avec succès `[Ins,ok]`, supprimer avec succès `[Del,ok]`, tester l'appartenance avec succès `[In,true]` et avec échec `[In,False]`. La matrice de commutativité est représentée en figure III.ag.

Dans un système typé, chaque objet put posséder en option un contrôle de concurrence défini au niveau de la classe. Les verrouillages sont alors délégués au contrôleur du type d'objet. Celui-ci laisse passer simultanément les verrouillages en modes d'opérations commutatives. Par exemple, un ensemble pourra être verrouillé simultanément par deux transactions en mode `[Ins,ok]`, ou en `[In,False]` et `[Del,ok]`. Le contrôleur bloque seulement les opérations non commutatives (ordonnancement).

Les reprises en cas de panne ou d'obtention d'un résultat invalide (non verrouillé par exemple) sont cependant difficiles. En effet, comme pour le verrouillage altruiste, le modèle est ouvert et permet de voir des données modifiées par des transactions non encore terminées. Il faut donc gérer la portée des transactions, par exemple sous forme de listes de transactions dépendantes. L'effet domino introduit ci-dessus survient : lors de la reprise d'une transaction, toutes les transactions dépendantes sont reprises.

	[Ins,ok]	[Del,ok]	[In,true]	[In,False]
[Ins,ok]	1	0	0	0
[Del,ok]	0	1	0	1
[In,true]	0	0	1	1
[In,False]	0	1	1	1

Figure III.ag : *Commutativité d'opérations sur ensemble*

Certains auteurs ont aussi considéré la commutativité en avant et la commutativité en arrière. Par exemple `[In,true]` et `[Insert,ok]` commutent en avant mais pas en arrière : si l'on exécute ces deux opérations à partir d'un état s sur lequel elles sont définies, on obtient bien le même résultat quel que soit l'ordre. Mais si l'on a l'exécution `[Insert,ok] [In,true]`, on n'est pas sur que `[In,true]` soit définie sur l'état initial (l'objet inséré peur être celui qui a permis le succès de l'opération `In`). Donc, on ne peut pas commuter lorsqu'on défait et refait une exécution. Tout cela complique els procédures de reprises et l'exploitation de la commutativité des opérations.

III.2.4.4.5 Verrouillage multiple :

Cette section décrit notre contribution dans le cadre de ce mémoire de magister. Nous définissons une nouvelle opération dite de verrouillage multiple. Elle doit être utilisée de prime abord par toute transaction nécessitant des verrous : tous les verrous dont a besoin la transaction sont demandés au moyen de l'opération LockM(G1,M1,……,Gn,Mn) , les (Gi)i=1..n sont les objets verrouillés et (Mi)i=1..n les modes de verrouillage respectifs ; les verrous sont relâchés avec l'opération Unlock(Gi).

Lors d'une demande de verrouillage , si au moins un des objets est verrouillé ou le mode est incompatible , la transaction demandante est mise en attente jusqu'à la libération de tous les objets demandés par la transaction (ou jusqu'à ce que les modes soient compatibles).

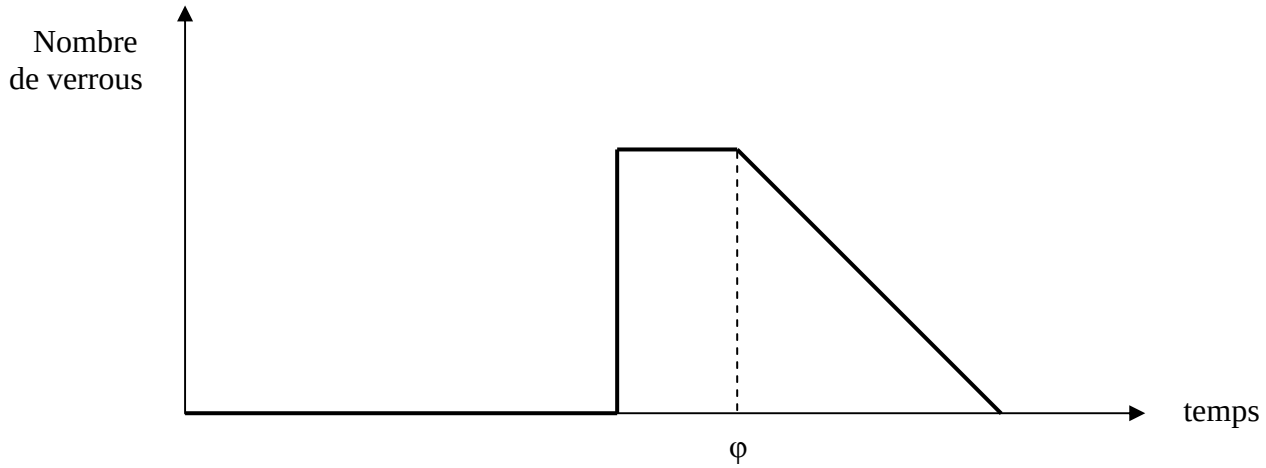


Fig III.ah : *comportement des transactions deux phases avec verrouillage multiple*

De plus cette manière de faire, et comme cela est décrit au paragraphe III.2.4.2.4 b), permet de prévenir les interblocages : on s'est attaqué à la condition de détention et d'attente.

Exemple d'utilisation :

T1	T2
LockM(x,R,y,W)	LockM(y,R,x,W)
lire(x)	lire(y)
écrire(y)	écrire(x)
Unlock(x)	Unlock(y)
Unlock(y)	Unlock(x)

Fig III.ai : *situation de non-interblocage entre T1 et T2*

La première transaction qui fait LockM obtient les deux verrous et continue son exécution ; durant cette exécution si l'autre transaction demande les verrous , elle est bloquée jusqu'à ce que la première relâche les deux verrous.

Algorithme de verrouillage et déverrouillage :

Constantes : n = nombre de transactions ; m = nombre de verrous

Structures de données :

```
ma : tableau [1..n,1..m] de (R,W,F)
      /* matrice des attentes initialisée à F */
      /* R : lecture , W : écriture , F : libre */
mv : tableau [1..n,1..m] de (R,W,F)
     /* matrice des verrous obtenus initialisée à F */
```

Algorithmes :

```
procédure LockM(G1,M1,……,Gp,Mp)   /* l ≤ p */
début  /* tc = n° transaction courante */
 α : Si ∀ i=1..p ( mv[j,Gi]=F ou compatible(mv[j,Gi],Mi)
                   pour j=1..n et j ≠ tc ) alors
       mv[tc,Gk] := Mk pour k=1..p
       ma[tc,Gk] := F pour k=1..p
     sinon
       ma[tc,Gk] := Mk pour k=1..p
       bloquer la transaction tc
       Aller_à α  /* au réveil */
     Finsi
Fin  /* de la procédure LockM */

procédure Unlock(G)
début
  mv[tc,G] := F
  pour j:=1 à n faire
    si ma[j,G] ≠ F alors
      ma[j,G] := F
      Réveiller la transaction j
    Finsi
  Finpour
Fin  /* de la procédure Unlock */
```

Figure III.aj : *Algorithmes de verrouillage et déverrouillage*

Remarque : Il n'existe pas de situations où une transaction détient quelques verrous et en attente d'autres : soit elle possède tous les verrous demandés, soit en attente des verrous demandés.

Il s'en suit que si l'on suppose qu'il y a interblocage, alors toute transaction du groupe est en attente donc il existe au moins un verrou détenu par une des transactions ce qui est une contradiction.

III.2.5 Contrôle de concurrence optimiste : [8]

Le verrouillage est une solution pessimiste : il empêche les conflits de se produire, ou plutôt les transforme en verrou mortel. Analysons maintenant une autre gamme de solutions qualifiées d'optimiste qui laissent se produire les conflits et les résout ensuite.

III.2.5.1 Ordonnancement par estampillage : [8]

Bien que le verrouillage avec prévention ou détection du verrou mortel soit la technique généralement appliquée dans les SGBD, de nombreuses autres techniques ont été proposées. En particulier l'**ordonnancement par estampille** peut être utilisé non seulement pour résoudre les verrous mortels comme vu ci-dessus, mais plus complètement pour garantir la sérialisabilité des transactions.

Une méthode simple consiste à conserver pour chaque objet accédé (tuple ou page), l'estampille du dernier écrivain W et celle du plus jeune lecteur R. Le contrôleur de concurrence vérifie alors :

1. que les accès en écriture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc l'écrivain W et le lecteur R.
2. que les accès en lecture s'effectuant dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc par rapport à l'écrivain W.

On abouti donc à un contrôle très simple d'ordonnancement des accès conformément à l'ordre de lancement des transactions. En cas de désordre, il suffit de reprendre la transaction ayant créé le désordre. Les contrôles nécessaires en lecture et écriture sont résumés en figure III.ak.

```
/* Contrôle d'ordonnancement des transactions */
Fonction Ecrire (Ti,O) /* la transaction Ti demande l'écriture de O */
Début
  Si ts(Ti) < W(O) ou ts(Ti)<R(O) alors abort(Ti)
  sinon executer_ecrire(Ti,O)
  finsi
Fin
```

Figure III.ak (1ere partie) : *Algorithme d'ordonnancement des accès par estampillage* [8]

```
Fonction Lire (Ti,O) /* la transaction Ti demande la lecture de O */
Début
  Si ts(Ti) < W(O) alors abort(Ti)
  sinon executer_lire(Ti,O)
  finsi
Fin
```

Figure III.ak : *Algorithme d'ordonnancement des accès par estampillage* [8]

L'algorithme d'ordonnancement par estampillage soulève plusieurs problèmes. De fait, les estampilles W et R associés à chaque objet remplacent les verrous. Il n'y a pas d'attente, celles-ci étant remplacées par des reprises de transaction en cas d'accès ne respectant pas l'ordre de lancement des transactions. Ceci conduit en général à beaucoup trop de reprises. Une amélioration possible consiste à garder d'anciennes versions des objets. Si l'estampille du lecteur ne dépasse pas celle du dernier écrivain, on peut délivrer une ancienne version, plus exactement, la première inférieure à l'estampille du lecteur. Ainsi, il n'y a plus de reprises lors des lectures. La méthode est cependant difficile à mettre en œuvre et n'est guère utilisée aujourd'hui.

III.2.5.2 Certification optimiste : [8]

La certification optimiste est une méthode de type curative, qui laisse les transactions s'exécuter et effectue un contrôle garantissant la sérialisabilité en fin de transaction. Une transaction est divisée en trois phases : phase d'accès, phase de certification et phase d'écriture. Pendant la phase d'accès, chaque contrôleur de concurrence garde les références des objets lus/écrits par la transaction. Pendant la phase de certification, le contrôleur vérifie l'absence de conflits avec les transactions certifiées pendant la phase d'accès. S'il y a conflit, la certification est refusée et la transaction défaite puis reprise. La phase d'écriture permet l'enregistrement des mises à jour dans la base pour les seules transactions certifiées.

En résumé, on introduit ainsi la notion de certification qui peut être effectuée de différentes manières :

Définition : **Certification de transaction** (*Transaction certification*)

Action consistant à vérifier et garantir que l'intégration dans la base de données des mises à jour préparées en mémoire par une transaction préservera la sérialisabilité des transactions.

Vérifier l'absence de conflits pourrait s'effectuer en testant la non-introduction de circuits dans le graphe de précédence. L'algorithme commun de certification est plus simple. Il consiste à mémoriser les objets lus (*Read Set RS*) et écrits (*Write Set WS*) par une transaction. La certification de la transaction Ti consiste

à tester que RS(Ti) n'intersecte pas avec WS(Tj) et que WS(Ti) n'intersecte pas avec WS(Tj) ou RS(Tj) pour toutes les transactions Tj lancées après Ti. On vérifie donc que les transactions n'agissent pas en modes incompatibles avec les transactions concurrentes avant de les valider. L'algorithme est présenté n figure III.al.

En résumé, cette méthode optimiste est analogue au verrouillage, mais tous les verrous sont laissés passant et les conflits ne sont détectés que lors de la validation des transactions. L'avantage est la simplicité du contrôleur de concurrence qui se résume à mémoriser les objets accédés et à un test simple d'intersection d'ensembles de référence lors de la validation. L'inconvénient majeur est la tendance à reprendre beaucoup de transactions en cas de conflits fréquents. La méthode optimiste est donc seulement valable pour les cas où les conflits sont rares.

```
Fonction Certifier(Ti) : booléen
Début
  Certifier := VRAI
  Pour chaque transaction t concurrente faire
    Si RS(Ti) ∩ WS(t) ≠ Ø ou WS(Ti) ∩ RS(t) ≠ Ø
       ou WS(Ti) ∩ WS(t) ≠ Ø alors
      Certifier := FAUX
      Abort(Ti)
    finsi
  FinPour
Fin
```

Figure III.al : *Algorithme de certification optimiste* [8]

III.2.5.3 Estampillage multi-versions : [8]

Comme pour le verrouillage deux phases et même mieux, la stratégie d'ordonnancement par estampillage vue ci-dessus peut être améliorée en gardant plusieurs versions d'un même granule. Pour chaque objet O, le système peut maintenir :

1. un ensemble d'estampilles en écriture {EEi(O)} avec les valeurs associées {Oi}, chacune d'elles correspond à une version i ;
2. un ensemble d'estampilles en lecture {ELi(O)}

Il est alors possible d'assurer l'ordonnancement des lectures par rapport aux écritures sans jamais reprendre un transaction lisant. Pour cela, il suffit de délivrer à une transaction Ti demandant à lire l'objet O la version ayant une estampille en écriture immédiatement inférieure à . Ainsi, Ti précédera toutes les créations d'estampilles supérieures écrivant l'objet considéré et suivra celles d'estampilles inférieures.

Ti sera donc correctement séquencée. Tout se passe comme si Tj avait demandé la lecture juste après l'écriture de la version d'estampille immédiatement inférieure. L'algorithme de contrôle pour l'opération LIRE avec un dispositif d'ordonnancement partiel multi-versions est représenté en figure III.am.

```
/* Lecture de la bonne version dans le passé */
Fonction Lire (Ti,O) /* la transaction Ti demande la lecture de l'objet O */
Début
  j := index de la dernière version de O
  Tant que ts(Ti) < W(O) faire
    j := j-1  /* chercher la version avant Ti */
  finTantque
  executer_lire(Ti,Oj)  /* lire la bonne version */
Fin
```

Figure III.am : *Algorithme de lecture avec ordonnancement multi-versions* [8]

Il est en général très difficile de refaire le passé. Cependant, il est parfois possible de forcer l'ordonnancement des écritures de Ti en insérant une nouvelle version créée par Ti juste après celle d'estampille immédiatement inférieure, soit Oj. Malheureusement , si une transaction Tk (k>i) a lu la version Oj, alors cette lecture doit aussi être reséquencée. Ce n'est possible que si la transaction Tk pouvait être reprise. Afin d'éviter la reprise de transactions terminées, on préférera reprendre l'écrivain Tj avec une nouvelle estampille i' supérieure à k.
L'algorithme de contrôle de l'opération WRITE correspondant est représenté figure III.an. Les notations sont identiques à celles utilisées ci-dessus, les indices désignant les numéros de versions d'objets.

```
/* Réordonnancement des écritures dans le passé */
Fonction Ecrire (Ti,O) /* la transaction Ti demande l'écriture de O */
Début
  j := index de la dernière version de O
  Tant que ts(Ti) < W(O) faire
    j := j-1  /* chercher la version avant Ti */
  finTantque
  si ts(Ti) < R(Oj) alors abort(Ti) finsi
                        /* abort si lecture non dans l'ordre */
  executer_ecrire(Ti,Oj)  /* écrire en bonne place */
Fin
```

Figure III.an : *Algorithme d'écriture avec ordonnancement multi-versions* [8]

La figure III.ao illustre l'algorithme.

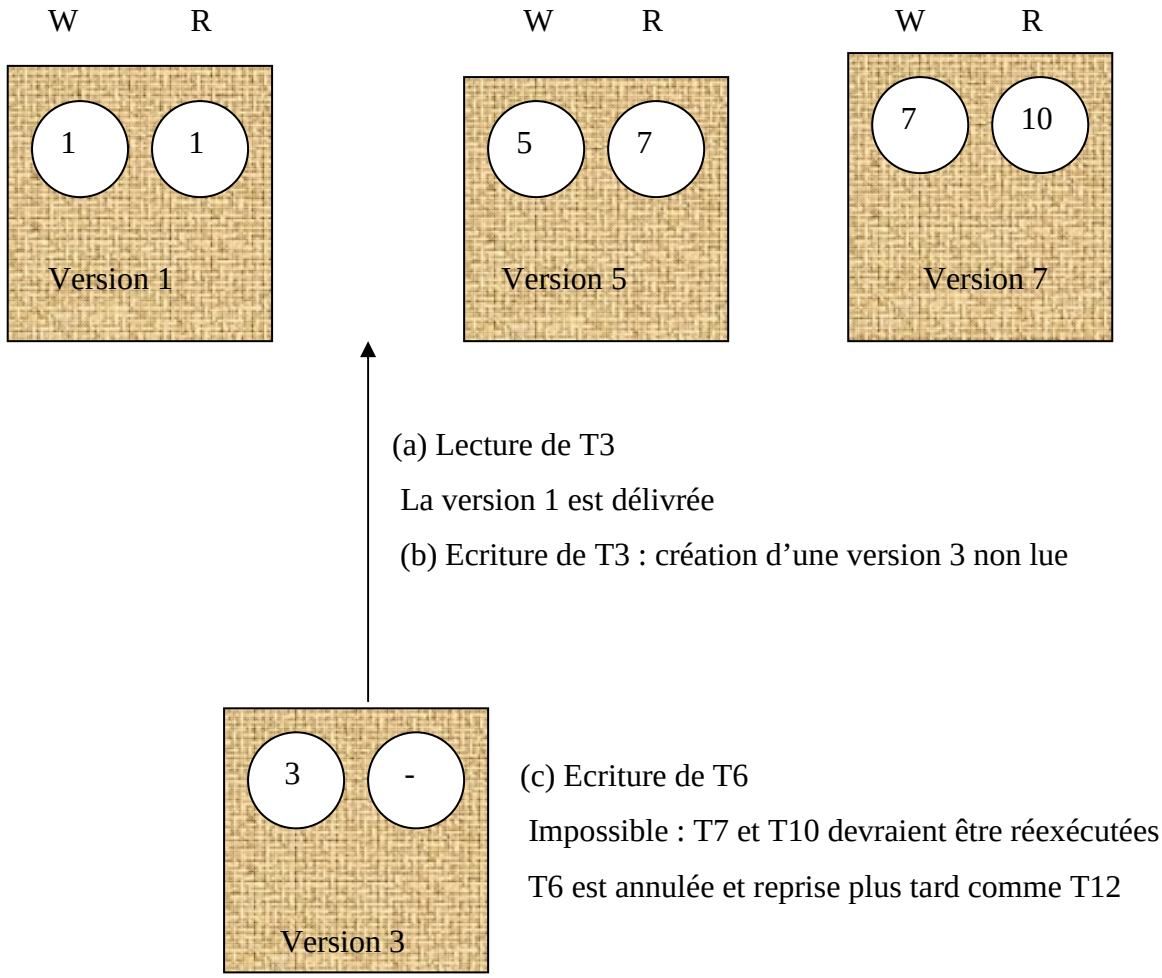


Figure III.ao : *Exemple de reséquencement*

Les transactions entrent en conflits sur un objet O unique dont les versions successives sont représentées par des rectangles. La situation originale est représentée en haut de la figure. Trois versions de l'objet existent, successivement créées par les transactions 1, 5 et 7. La version 1 a été lue par la transaction 1, la version 5 par la transaction 7 et la version 7 par la transaction 10. Nous supposons que T3 accomplit une écriture sur l'objet O après l'avoir lu. La nouvelle version 3 créée est insérée en bonne place. Nous supposons ensuite que T6 procède à une écriture sur O. L'objet ayant été lu par T7, il faudrait refaire le passé. On préférera annuler T6 et la relancer plus tard.

En résumé, beaucoup d'algorithmes basés sur des estampilles peuvent être inventés pour contrôler les accès concurrents. Il est de même possible de mixer estampilles et verrouillage, comme déjà vu au niveau des algorithmes DIE-WAIT et WOUND-WAIT. Cependant les performances de ces algorithmes restent faibles car ils provoquent tous des reprises qui deviennent de plus en plus fréquentes lorsqu'il y a un plus grand nombre de conflits, donc lorsque le système est chargé. Voilà sans doute pourquoi la plupart des SGBD utilisent le verrouillage deux phases.

III.2.6 Reprise sur panne : [17], [8], [3]

Sur un système informatique, nous pouvons définir plusieurs types de « pannes », qui impliquent plusieurs problèmes différents :

- Abandon d'une transaction : cela peut impliquer de défaire les actions de la transaction abandonnée. Ce cas de « panne », dans les systèmes distribués traditionnels, peut être par exemple dû à un problème d'accès concurrent sur une donnée, ou à l'utilisateur [17].
- Panne su système : lorsque le système tombe complètement en panne, le contenu de la RAM est perdu, et le contenu du support de mémoire stable n'est pas affecté par ce type de panne. Dans ce cas, il faut complètement défaire toutes les transactions en cours (cf. cas précédent), ou éventuellement les transactions validées (dont les effets ne sont pas présents sur disque).
- Panne de journal : dans ce cas là, toutes les modifications effectuées sont perdues. Cela implique de tout refaire depuis le début ou d'utiliser un support de sauvegarde.

De manière à implanter les propriétés d'atomicité et de durabilité, nous devons utiliser des mécanismes de reprise sur panne [2], [17].

Dans cette partie, nous allons tout d'abord étudier les différentes gestions d'écriture possibles (mise à jour des données directement sur le support ou non) puis nous étudierons différentes techniques de reprise sur panne (utilisation de journaux ou de mécanismes de répliquât).

III.2.6.1 Influence de la gestion du cache sur la reprise :

Il s'agit de la gestion du cache et de la façon dont il est recopié sur le support stable. Il existe quatre méthodes de mises à jour des données [9] :

- Soit les objets présents dans le cache sont recopié avant la validation de la transaction. Il se peut alors que l'on soit obligé de *défaire*[3] des transactions, mais on a jamais à *refaire*[4] une transaction.
- Soit les données sont mises à jour uniquement après validation (dans ce cas la transaction n'est jamais défaite, mais il peut être nécessaire de la refaire).
- Soit la mise à jour des données sur le support est sans contrainte, ce qui est le cas le plus fréquent. On peut alors avoir à défaire et à refaire des actions de transactions.
- Enfin la dernière méthode consiste à mettre à jour les données non pas en remplaçant les données initiales, mais en faisant les mises à jour « ailleurs » sur le support stable.

Les mises à jour des données sont appelées des ***points de reprise***. Ce sont en fait des points de journal, où on est assuré que toutes les actions antérieures ont leur effets présents sur le support stable.

3. On appelle « défaire » une transaction, éliminer les actions d'une transaction sur le disque
4. On appelle « refaire » une transaction, rejouer une transaction pour intégrer les actions perdues

III.2.6.2 Méthodes de Journalisation :

Les méthodes de journalisation (« *Logging* » en anglais) sont basées sur le principe qu'une transaction est en fait une suite d'actions. A chaque action, il suffit de laisser une trace dans un journal. Alors pour refaire ou défaire une transaction, on parcourt le journal.

Il existe plusieurs types de journaux [27] pour assurer le recouvrement des actions d'une transaction :

- Les Journaux de valeurs : Dans ces journaux, on stocke la valeur des objets modifiés par une transaction. Il existe deux types de journaux de valeurs, les *journaux des images avant* et les *journaux des images après*. Dans le premier cas, on sauvegarde une image des objets modifiés par la transaction avant que celle-ci ne change ; alors que dans le second cas, on sauvegarde la nouvelle valeur de ces objets.
- Les journaux différentiels : Dans ces journaux, on effectue un XOR (OU exclusif) entre ancienne et nouvelle valeur de l'objet. Alors le journal ne contient pas de réelles valeurs de données, ce qui permet de ne pas sortir de secret dans le journal.
- Les journaux d'opérations : Ce journal sauvegarde toutes les opérations effectuées sur les objets avec leurs opérandes et, nécessairement les opérations inverses correspondantes.

Quel que soit le type de journal, l'implémentation est en fait un fichier séquentiel (logiquement, et physiquement pour accélérer le débit des écritures) où est enregistré une suite d'actions. Ce fichier doit obligatoirement être stocké sur un support stable, car son contenu ne doit pas être perdu.

Pour défaire une transaction, nous avons besoin de conserver dans les journaux les *images avant* des données (c'est-à-dire la valeur des données avant modification par la transaction). Pour pouvoir défaire une transaction, il faut que l'image avant de la donnée soit écrite dans le journal avant que la donnée ne soit modifiée (donc avant que la donnée migre du cache). A l'inverse, pour refaire une transaction, il faut conserver les *images après* des données, et dans ce cas, les informations doivent être stockées dans le journal avant l'ordre de validation.

III.2.6.3 Méthode des Pages Ombres : [17]

La méthode des pages ombres (« *Shadow Pages* » en anglais) est une méthode de mises à jour « ailleurs » (cf. figure III.ap) qui peut se faire au moment de la validation de la transaction (ainsi nous n'avons pas besoin de journaux pour refaire ou défaire les transactions.

Le principe de fonctionnement des pages ombres est le suivant :

- Avant la transaction, la table d'indirection virtuelle est divisée en deux parties : une table valide qui pointe sur les pages courantes, et une table ombre.

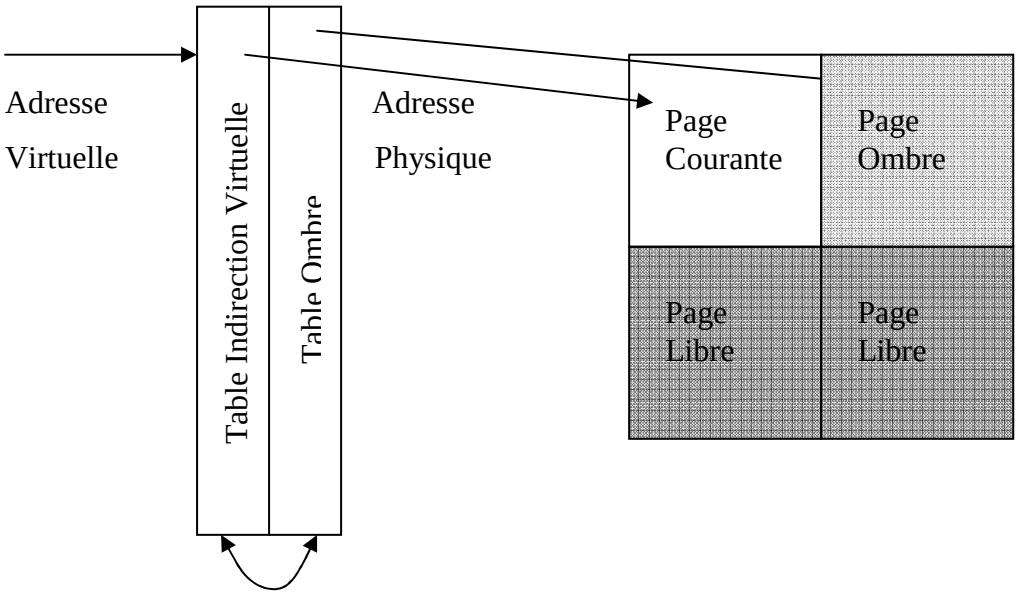


Figure III.ap : *Méthode des pages ombres*

- pendant la transaction, lors de la modification d'une donnée, la nouvelle valeur va être écrite « ailleurs » sur le support stable, et va être pointée par la partie ombre correspondante de la table d'indirection. Ainsi, les données de la page courante ne sont pas modifiées.
- A la validation, la nouvelle table valide est reconstituée : les parties de la table ombre pointant sur des données modifiées par la transaction font dorénavant partie de la table courante, et les parties contenant les données initiales sont maintenant dans la table ombre.
- En cas d'annulation, la table d'indirection virtuelle valide n'est pas modifiée (les données pointées par la page ombre sont donc perdues).

Ce mécanisme est très simple d'utilisation et d'implantation pour un système exécutant très peu de transactions simultanément. Il devient moins efficace que la journalisation pour les systèmes transactionnels plus évolués.

III.2.7 Validation de transaction : [8], [17], [3]

La validation de transaction doit intégrer toutes les mises à jour d'une transaction dans une base de données de manière atomique, c'est-à-dire que toutes les mises à jour doivent être intégrées ou qu'aucune ne doit l'être. L'atomicité de la validation rend les procédures d'annulation de transactions non validées simples. Le problème est donc de réaliser cette atomicité. Plusieurs techniques ont été introduites dans les systèmes afin de réaliser une validation atomique. Elles peuvent être combinées afin d'améliorer la fiabilité [11]. La plus part des SGBD combinent d'ailleurs les écritures en place et la validation en deux étapes.

III.2.7.1 Ecriture en place :

Avec cette approche, les écritures sont effectuées directement dans les pages de la base contenant les enregistrement modifiés. Elles sont reportées dans la base au fur et à mesure de l'exécution des commandes Insert, Update et Delete des transactions. Ce report peut être différé mais doit être effectué avant la validation de la transaction. Comme vu ci-dessus, les mises à jour sont écrites dans le journal avant d'être mises en place dans la base de donnée. L'atomicité de la validation d'une transaction est réalisée par écriture d'un enregistrement dans le journal [30]. Les écritures restent invisibles aux autres transactions tant qu'elles ne sont pas validées ; pour cela les pages ou les enregistrements sont verrouillés au fur et à mesure des écritures. La validation proprement dite consiste à écrire dans le journal un enregistrement « transaction validée ». Ensuite, les mises à jour sont rendues visibles aux autres transactions.

Dans tous les cas, la validation d'une transaction ayant mis à jour la base de données génère un nouvel état, ainsi des enregistrements dans le journal (images des pages modifiées et enregistrement transaction validée) comme indiqué en figure III.aq.

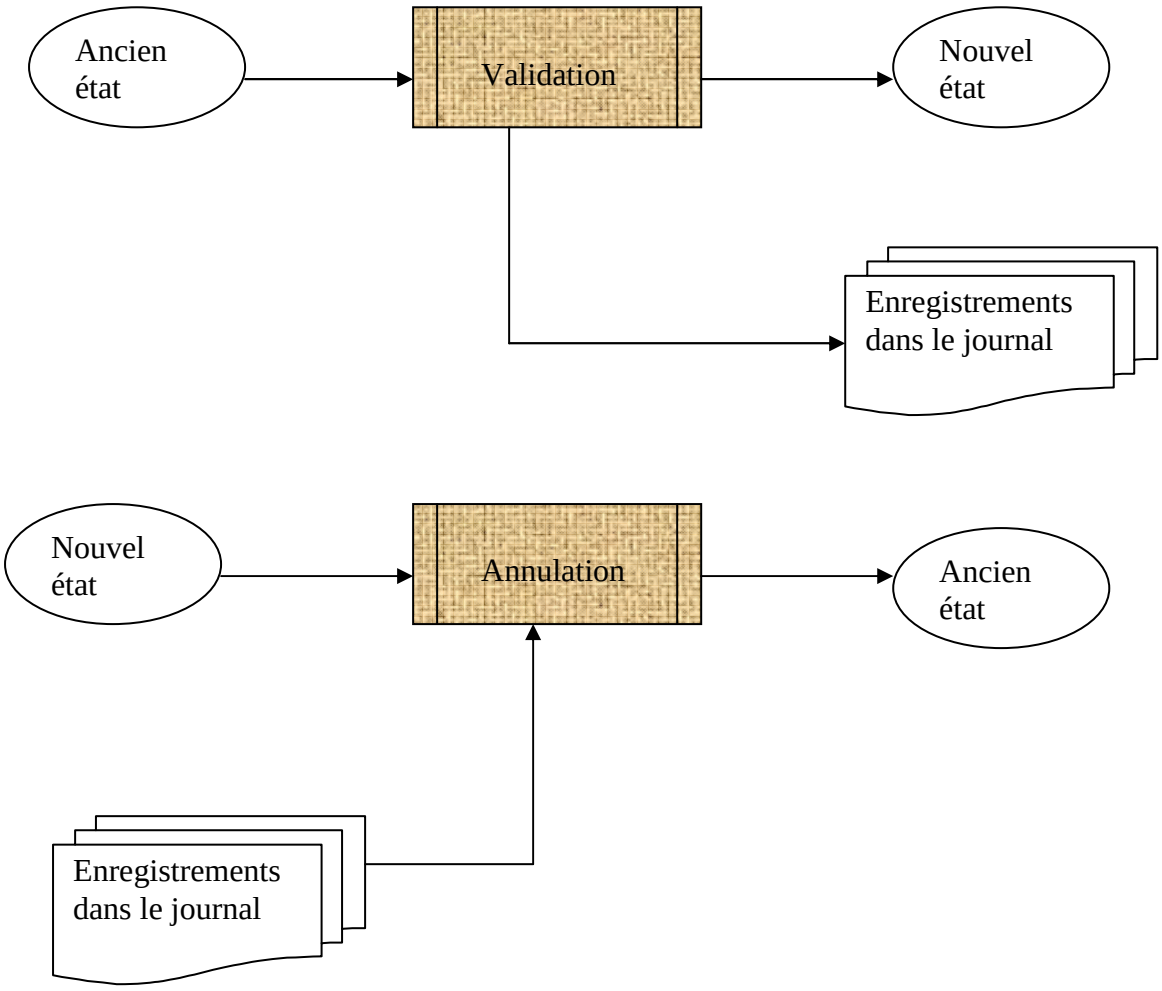


Figure III.aq : *Principes de la validation et de l'annulation avec journal*

L'annulation d'une transaction ayant mis en place des mises à jour dans la base est une procédure difficile. Elle s'effectue à partir du journal des images avant. L'annulation nécessite le parcours du journal à l'envers, c'est-à-dire en reculant dans le temps.

III.2.7.2 Ecriture non en place :

Avec cette approche, les pages modifiées seront recopiées dans de nouvelles pages en mémoire et écrites dans de nouveaux emplacements dans la base. L'atomicité de la validation d'une transaction peut être réalisée par la **technique des pages ombre** [21]. Comme vu précédemment, les pages nouvelles séparées et attachées à la transaction modifiante sont appelées pages différentielles. Les pages anciennes constituent les pages ombre. Avant toute lecture, il faut que le SGBD consulte les pages différentielles validées et non encore physiquement intégrées à la base. Cela conduit à alourdir les procédures de consultation et accroître les temps de réponse.

Une solution plus efficace a été proposée dans [20]. Elle consiste à réaliser une intégration physique atomique par **basculement des tables des pages**. Pour cela chaque fichier de la base de données possède un descripteur pointant sur la table des pages du fichier. Quand une transaction désire modifier un fichier, une copie de la table des pages est effectuée et les adresses des pages modifiées sont positionnées à leurs nouvelles valeurs, de sorte que la copie de la table des pages pointe sur la nouvelle version du fichier (anciennes pages non modifiées et nouvelles pages modifiées). La validation consiste alors simplement à mettre à jour le descripteur du fichier en changeant l'adresse de la table des pages.

III.2.7.3 Validation en deux étapes :

Dans la plupart des systèmes, un ensemble de processus collabore à la vie d'une transaction. Afin de coordonner la décision de valider une transaction, un **protocole de validation en deux étapes** est généralement utilisé. Ce protocole a été proposé dans un contexte de système réparti [20], [10] mais est aussi utile dans un contexte centralisé multiprocessus. La validation en deux étapes peut être perçue comme une méthode de gestion des journaux. Lors de la première étape, les images avant et après sont enregistrées dans le journal si bien qu'il devient ensuite possible de valider ou d'annuler la transaction quoi qu'il advienne (excepté une perte du journal) ; cette étape est appelée préparation à la validation. La seconde étape est la réalisation de la validation ou de l'annulation atomique, selon que la première étape s'est bien ou mal passée.

Le protocole de validation en deux phases coordonne l'exécution des commandes COMMIT par tous les processus coopérant à l'exécution d'une transaction. Le principe consiste à diviser la validation en deux phases. La phase 1 réalise la préparation de l'écriture des résultats des mises à jour dans la BD et la centralisation du contrôle. La phase 2, réalisée seulement en cas de succès de la phase 1, intègre effectivement les résultats des mises à jour dans la BD répartie. Le contrôle du système est centralisé sous la direction d'un processus appelé **coordinateur**, les autres étant des participants.

Nous résumons ces concepts ci-dessous :

Définition : **Protocole de validation en deux étapes** (*Two Phase Commit*)

Protocole permettant de garantir l'atomicité des transactions dans un système multiprocessus ou réparti, composé d'une préparation de la validation et de la centralisation du contrôle, et d'une étape d'exécution.

Définition : **Coordinateur de Validation** (*Commit Coordinator*)

Processus d'un système multiprocessus ou réparti qui dirige le protocole en centralisant le contrôle.

Définition : **Participant à la validation** (*Commit Participant*)

Processus d'un système multiprocessus ou réparti qui exécute les mises à jour de la trans--action et obéit aux commandes de préparation, validation ou annulation du coordinateur.

Le protocole de validation en deux étapes découle des concepts précédents. Lors de l'étape 1, le coordinateur demande aux autres sites s'ils sont prêts à commettre leurs mises à jour par l'intermédiaire du message PREPARER (en anglais *PREPARE*). Si tous les participants répondent positivement (prêts), le message VALIDER (en anglais *COMMIT*) est diffusé : tous les participants effectuent leur validation sur ordre du coordinateur. Si un participant n'est pas prêt et répond négativement (KO) ou ne répond pas (*time out*), le coordinateur demande à tous les participants de défaire la transaction (message ANNULER, en anglais *ABORT*).

Le protocole nécessite la journalisation des mises à jour préparées et des états des transactions dans un journal local à chaque participant, ceci afin d'être capable de retrouver l'état d'une transaction après unee éventuelle panne et continuer la validation. Le protocole est illustré een figure III.ar dans le cas favorable où un site client demande la validation à deux sites serveurs avec succès. Notez qu'après exécution de la demande de validation (VALIDER), les participants envoient un acquittement (FINI) au coordinateur afin de lui signaler que la transaction est maintenant terminée.

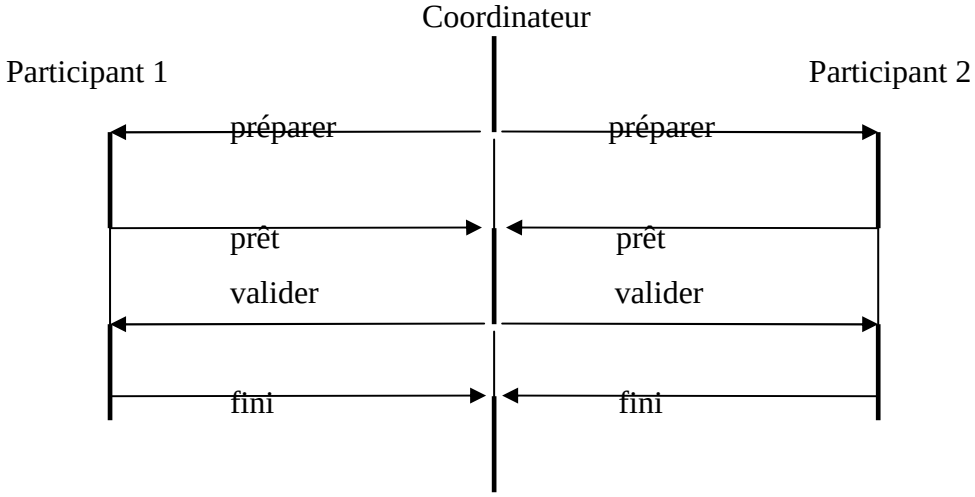


Figure III.ar : *validation en deux étapes avec succès* [8]

Le cas de la figure III.as est plus difficile : le participant 2 est en panne et ne peut donc répondre au message demande de préparation. Une absence de réponse est assimilée à un refus. Le coordinateur annule donc la transaction et envoie le message ANNULER. Le participant 1 annule la transaction. Le participant 2 l'annulera lorsqu'il repartira après la panne, une transaction non prête étant toujours annulée.

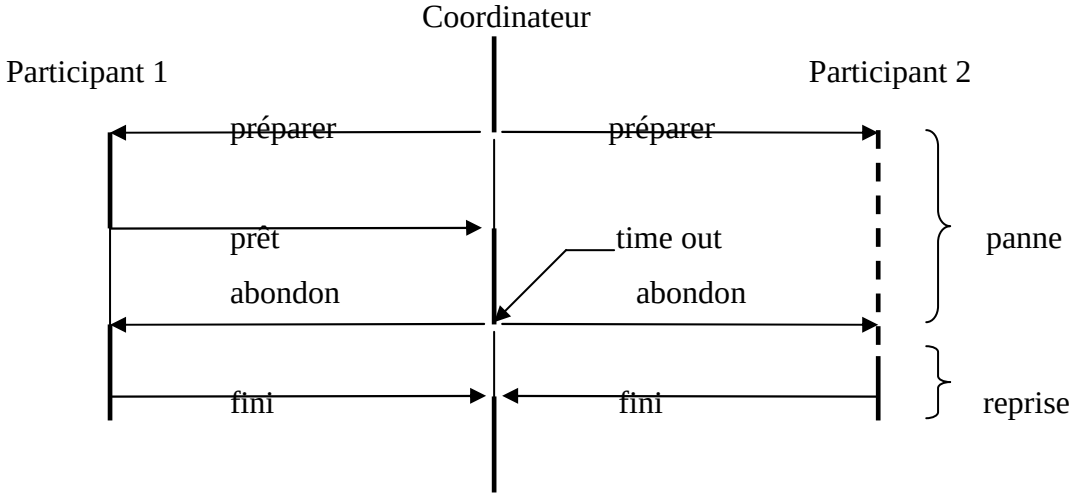


Figure III.as : *validation en deux étapes avec panne totale du participant 2* [8]

Le cas de la figure III.at est encore plus difficile : le participant 2 tombe en panne après avoir répondu positivement à la demande de préparation. Le coordinateur envoie donc le message COMMIT qui n'est pas reçu par le participant 2. Heureusement, celui-ci a dû sauvegardé l'état de la sous-transaction et ses mises à jour dans un journal fiable sur disque. Lors de la procédure de reprise, le journal sera examiné. La transaction étant détectée prête à commettre, son état sera demandé au coordinateur (ou à un participant quelconque) par un message STATUS. En réponse à ce message, la validation sera confirmée et les mises à jour seront appliquées à partir du journal. Les deux sous-transactions seront donc finalement validées.

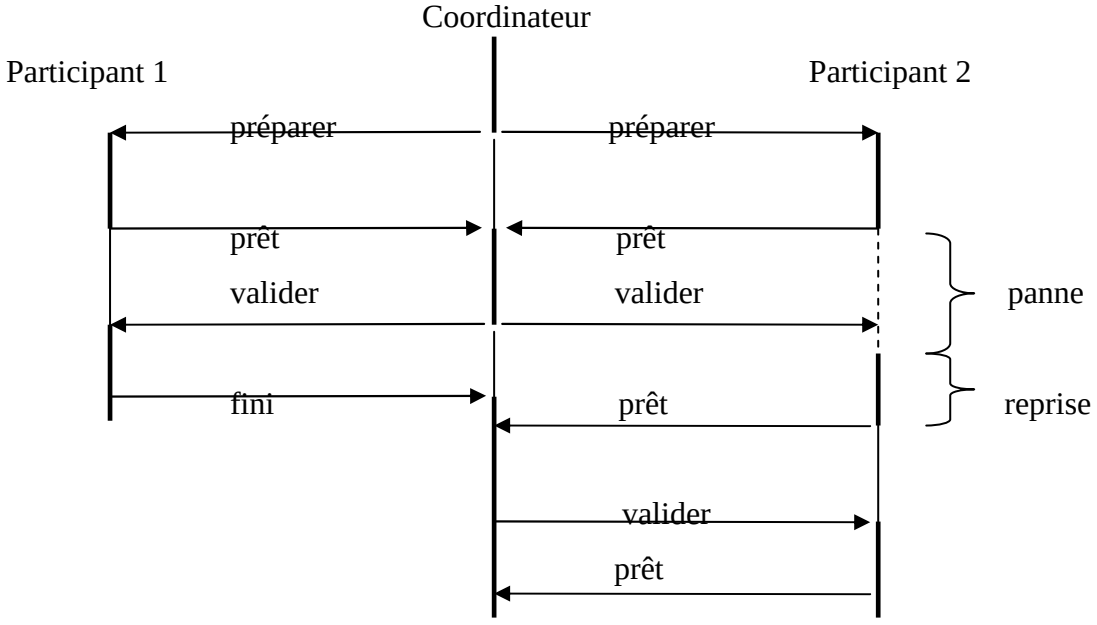


Figure III.at : *validation en deux étapes avec panne partielle du participant 2* [8]

Au-delà du protocole en deux étapes, il existe des protocoles en trois étapes qui évitent les blocages du protocole précédent en cas de panne du coordinateur. Le plus courant est le protocole dit d'*abort* supposé, qui en cas de panne du coordinateur suppose l'abandon de la transaction. Ceci est même possible en deux phases [22].

Le protocole en deux étapes a été standardisé par l'ISO. TP est le protocole standard proposé par l'ISO dans le cadre de l'architecture OSI afin d'assurer une validation cohérente des transactions dans un système distribué. Le protocole est arborescent en ce sens que tout participant peut déclencher une sous-transaction distribuée. Un site responsable de la validation de la sous-transaction est choisi. Un coordinateur est responsable de ses participants pour la phase 1 et collecte les PREPARE. Il demande ensuite la validation ou l'*abort* selon la décision globale à un site appelé ***point de validation*** qui est responsable de la phase 2 : c'est le point de validation qui envoie les COMMIT aux participants. L'intérêt du protocole, outre l'aspect hiérarchique, est que le point de validation peut être différent du coordinateur : ce peut être un nœud critique dans le réseau dont la présence à la validation est nécessaire. La figure III.au illustre ce type de protocole hiérarchique avec point de validation.

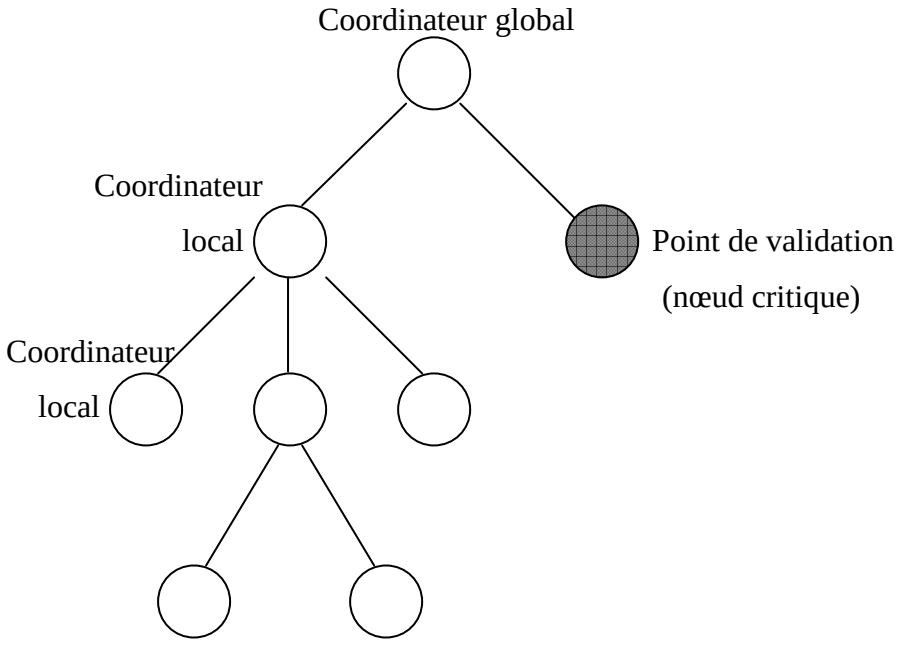


Figure III.au : *le protocole hiérarchique avec point de validation TP*

III.2.8 Normes et Services Transactionnels existants : [17]

Il existe plusieurs standards et normes pour définir les modèles transactionnels. Les standards les plus couramment utilisés sont ceux définis par les consortiums (X/OPEN ou OMG) ou des constructeurs en position de quasi-monopole (Microsoft).

III.2.8.1 Le protocole OSI TP :

OSI TP[5] [15] est le protocole que nous avons cité au paragraphe précédent. C'est un protocole transactionnel conforme au protocole de validation à deux phases dans un système distribué. Ce protocole fournit à la fois des services transactionnels (tels que la validation ou l'annulation de transaction), mais aussi un protocole de communication utilisé pour véhiculer les ordres transactionnels au travers du réseau.

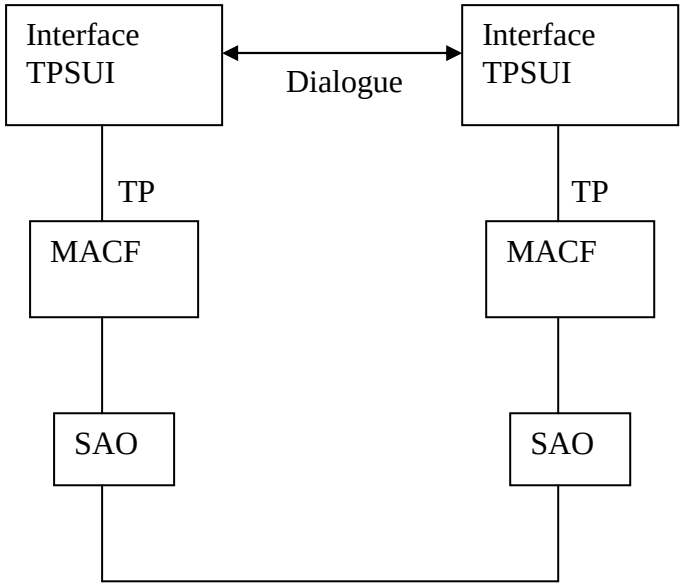


Figure III.av : *le modèle OSI-TP*

Le MACF[6] est le noyau de la machine OSI-TP, qui offre différents services aux utilisateurs TPSUI[7] (ces services sont préfixés de TP : par exemple TP-BeginTransaction). Il coordonne aussi les interactions avec les subordonnées (SAO[8]) qui encodent et décodent les informations transmises entre deux MACF.

Nous ne décrirons pas plus ici les services OSI-TP. Cependant il est à noter que ce protocole a deux grands avantages :

- il est standardisé : ce système permet à des systèmes hétérogènes d'interopérer pour le bon déroulement de la transaction distribuée.
- Il est ouvert : le composant clé est le MACF, mais le TPSUI et le SAO peuvent être considérés comme des interfaces avec les utilisateurs et le réseau, qui cachent le MACF, et le détachent de l'environnement d'exécution.

5. OSI-TP : Open System Interconnection Transaction Processing
6. Multiple Association Control Function
7. TPSUI : Transaction Processing Service User Invocation
8. SAO : Single Association Object

III.2.8.2 Le modèle DTP de X/OPEN :

Le modèle DTP[9] de X/OPEN[10] est un standard d'interfaces entre composants transactionnels [17].

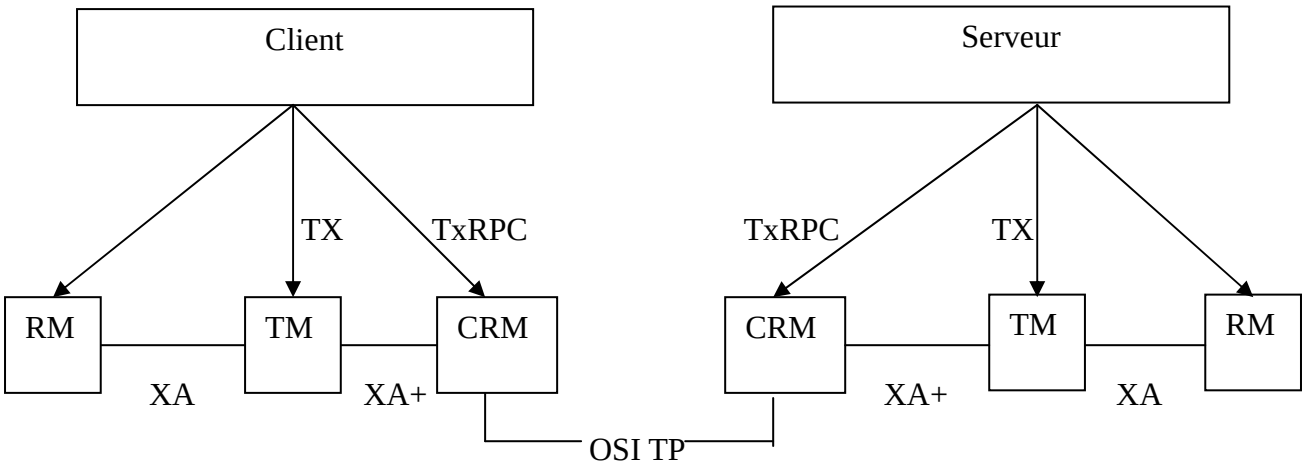


Figure III.aw : *Le modèle DTP de X/OPEN*

Les différents composants du modèle DTP (cf. figure III.aw) sont :

- l'AP (Application Program) qui est l'application qui définit les actions de la transaction
- le RM (Ressource Manager) qui est le gestionnaire de ressources de l'environnement de l'application. Il gère l'accès aux données.
- le TM (Transaction Manager) qui est le gestionnaire de la transaction. Il assure les services de coordination pour les transactions (gestion des identifiants, validation, etc.)
- le CRM (Communication Ressource Manager) qui gère les communications entre des applications distribuées sur plusieurs domaines (c'est-à-dire dépendant de plusieurs gestionnaires de transaction).

Les services de communication sont ceux définis par OSI-TP.

III.2.8.3 OTS de l'OMG :

L'OMG a défini un service transactionnel (OTS[11]) [17] pour les applications distribuées basées sur une architecture CORBA.

Le principe de fonctionnement est le suivant :

- Le client, qui est l'initiateur de la transaction, s'enregistre auprès de l'OTS, et crée une nouvelle transaction. Pour cela, il a deux solutions. Il peut utiliser l'interface *current* (il utilise alors un mode appelé *indirect*, qui rend transparent la gestion du contexte transactionnel), ou bien il utilise directement l'objet *TransactionFactory* de l'OTS (cette méthode est dite *directe*), qui lui renvoie un identifiant de transaction.

9. DTP : Distributed Transaction Processing
10. X/OPEN : est un consortium d'industriels qui a défini des standards pour les transactions distribuées
11. OTS : Object Transaction Services

- Lorsque le serveur reçoit une requête de la part du client, il reçoit simultanément le contexte de la transaction (soit de manière *explicite*, c'est-à-dire, en tant que paramètre de la requête, soit de manière implicite, c'est-à-dire véhiculé par l'ORB). Le serveur va alors s'enregistrer auprès de l'OTS.
- A la fin de la transaction, le client demande la validation (ou l'annulation), et l'OTS exécute alors un algorithme de validation à deux phases avec les serveurs.

Plusieurs OTS peuvent collaborer sur plusieurs ORB, via une technique appelée technique d'interposition [17].

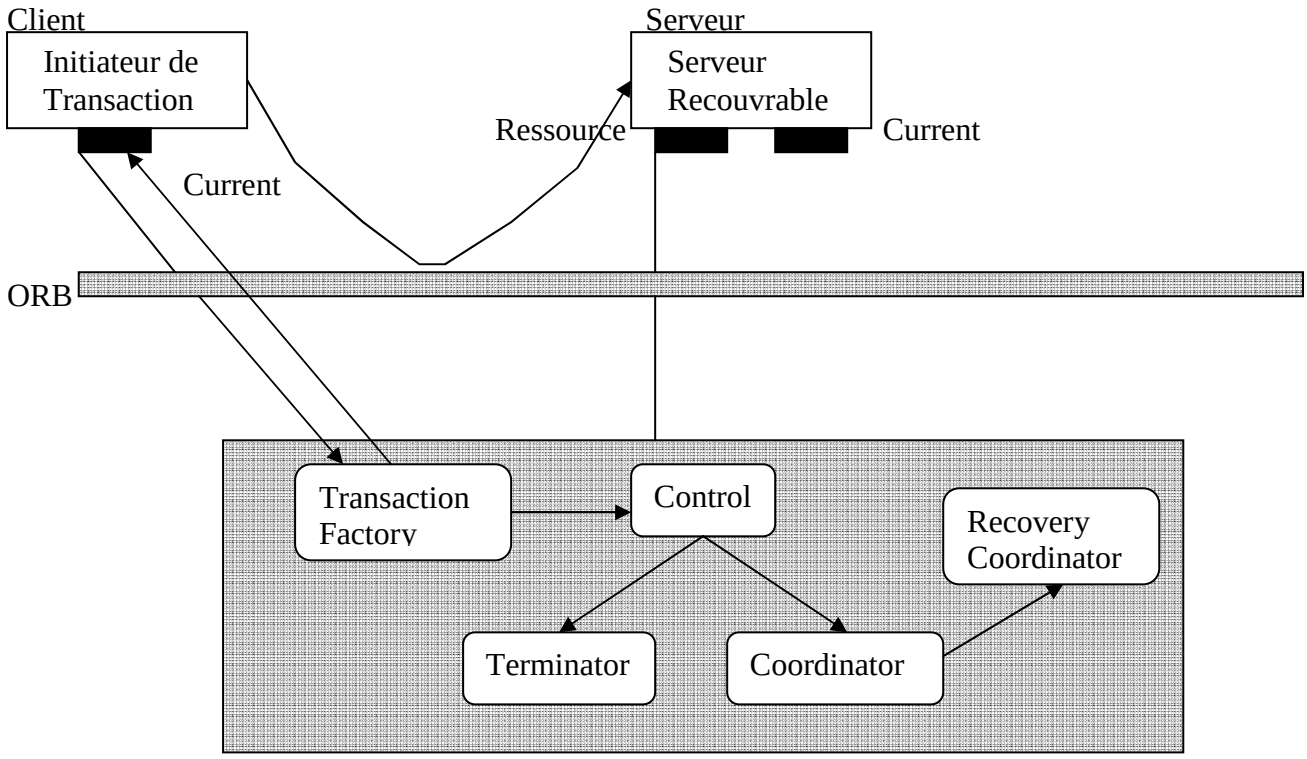


Figure III.ax : *Le système de transaction OTS*

III.2.8.4 MTS de Microsoft :

Le service transactionnel MTS[12] est basé sur une architecture 3-tiers.

Dans les architectures 3-tiers, le client, les traitements et les données sont séparés en trois composants qui peuvent être distants sur le réseau (cf. figure III.ay) :

- le composant de présentation d'interface, qui permet l'interaction avec l'utilisateur, et qui envoie des requêtes vers les services applicatifs (il peut être comparé à un client dans une architecture client / serveur),
- le composant applicatif, qui effectue les opérations logiques, ainsi que tous les traitements, et qui envoie des requêtes vers les bases de données,
- les serveurs de données, qui répondent aux requêtes du composant applicatif.

12. MTS : Microsoft Transaction Server

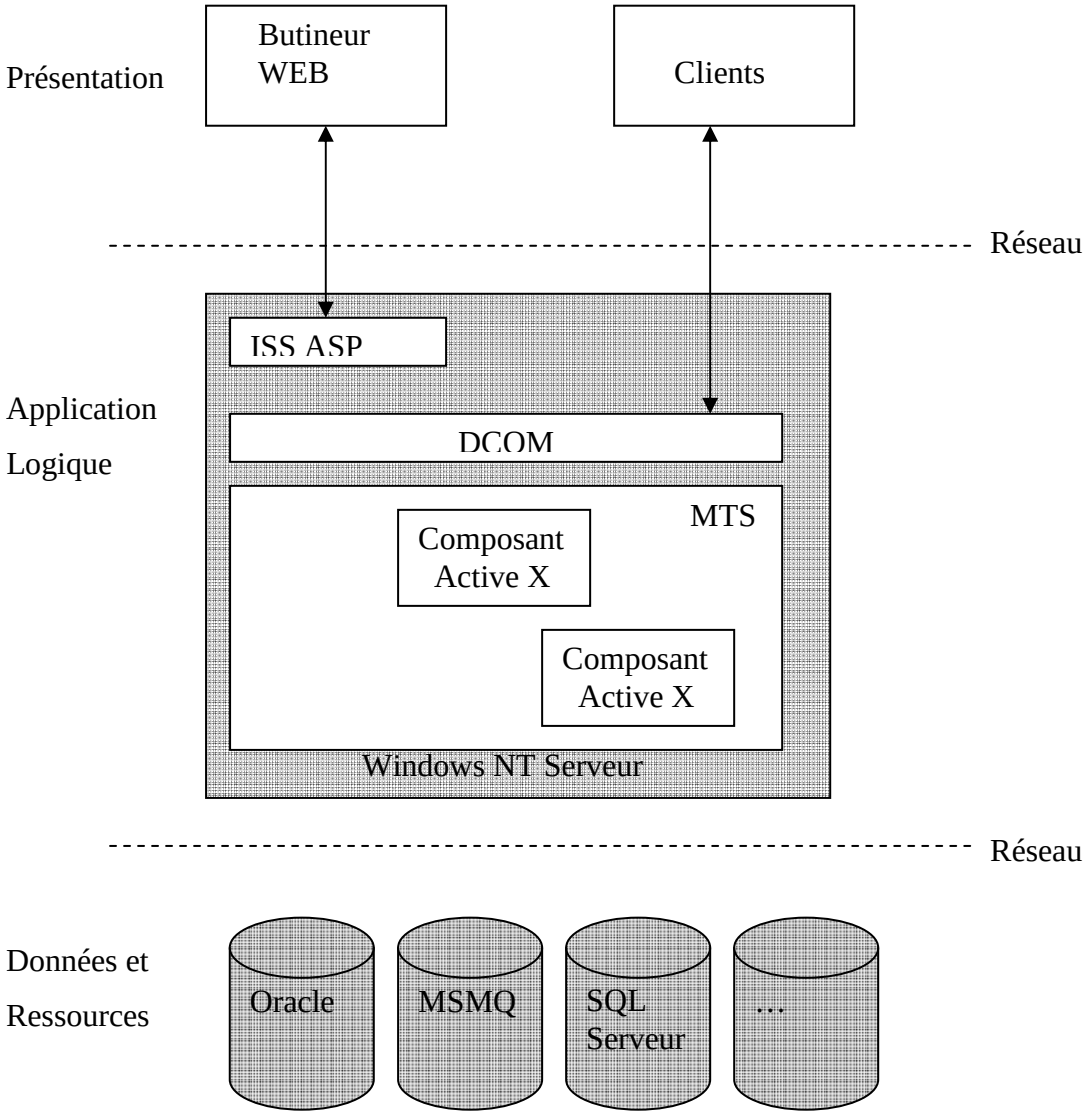


Figure III.ay : *MTS et les architectures 3-tiers*

Le service transactionnel de Microsoft est basé sur les architecture COM[13] et DCOM[14]. Ces architectures peuvent être comparées à CORBA, par les services qu'elles offrent.

Dans le cadre d'une application utilisant MTS, le composant applicatif s'exécute sous le contrôle du serveur transactionnel. Les clients qui invoquent ces services via DCOM, peuvent être soit des applications classiques, soit des scripts ASP[15] s'exécutant au travers d'un service Internet IIS[16] [17].

Le principal avantage de MTS est qu'il est une solution adaptée aux mécanismes développés par Microsoft au sein de ses systèmes d'exploitation.

13. COM : Component Object Model
14. DCOM : Distributed Component Object Model
15. ASP : Active Server Page
16. IIS : Microsoft Internet Information Server

III.3 Besoin transactionnel : [17]

Les nouvelles applications des cartes à microprocesseur posent de nombreux problèmes. A ces problèmes (propres aux applications) s'ajoutent ceux propres aux cartes. La solution envisagée pour pallier à cela est l'implantation dans la carte d'un gestionnaire de ressources, basé sur le modèle transactionnel [17].

La problématique des nouvelles applications[17] carte nous pousse à adopter un niveau transactionnel plus élevé que celui de la simple implantation d'un mécanisme de reprise sur panne minimal.

III.3.1 Atomicité intra-APDU :

Certaines cartes comme les Java Cards disposent des ordres « Begin_Transaction », « Abort_Transaction » et « Commit_Transaction ». Cependant il ne s'agit là que de rendre atomique l'exécution d'un petit nombre d'instructions à l'intérieur d'un ordre APDU.

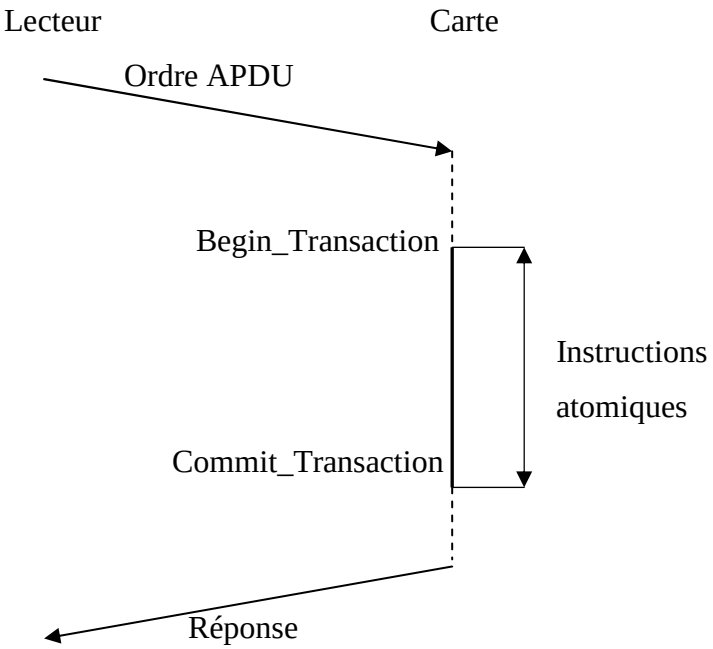


Figure III.az : Instructions atomiques intra-APDU [17]

Ce degré d'atomicité, qui vise à rendre atomique l'exécution d'instructions dans la carte, répond aux besoins actuels des applications. En effet, dans les applications carte actuelles, il n'y a pas d'accès concurrents aux données (car il y a une seule application par connexion), et la reprise sur panne s'effectue facilement (pas d'aspect distribué de l'application, et pas d'exécutions sur plusieurs sessions).

Cependant, ce niveau de transaction est insuffisant pour rendre tolérant aux pannes l'exécution des services carte, même les plus simples.

17. les nouvelles applications peuvent être des applications des cartes multi-services avec ou sans partage de données entre applications, des applications longues. On peut trouver également les applications multi-cartes …etc.

III.3.2 Atomicité APDU :

Ce degré d'atomicité, qui vise à rendre atomique l'exécution d'une APDU dans la carte, a comme principal intérêt de rendre la carte tolérante à l'arrachement (ce qui est la panne la plus fréquente). En effet lors d'un arrachement de la carte, le terminal ne reçoit pas ses mots d'états et considère que la carte n'a pas effectué le traitement de l'APDU.

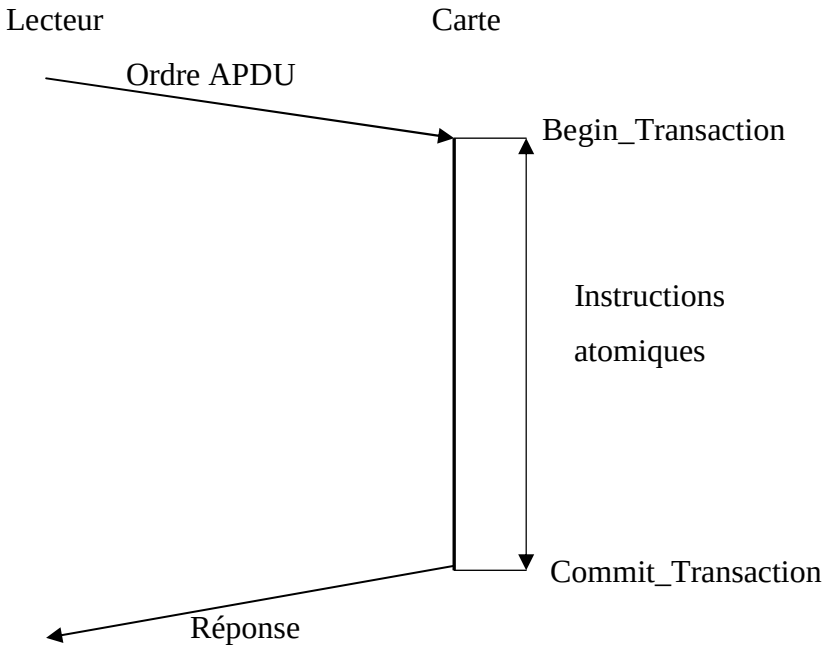


Figure III.ba : APDU exécutée atomiquement [17]

Le cas que nous venons de voir (atomicité intra-APDU) ne permet pas de garantir l'exécution totale, ou l'annulation totale en cas de panne, du traitement *complet* d'une APDU. En effet cela oblige le programmeur du service à placer correctement les ordres de « Begin_Transaction » et de « Commit_Transaction ». Dans l'état actuel des techniques utilisées dans les systèmes d'exploitation des cartes à microprocesseur, nous avons deux possibilités :

- soit la carte a un simple système d'écriture en mémoire non volatile très simple (c'est-à-dire qu'elle n'utilise pas de cache). Dans ce cas là, les modifications sur les données sont rendues persistantes au fur et à mesure de l'exécution des traitements (sauf dans le cas de la figure III.az, où l'on stocke les modifications d'abord dans un buffer).
- Ou bien la carte est munie d'un cache pour accélérer les écritures, et dans ce cas là, les effets sur les données sont rendus persistants au vidage du cache en mémoire non volatile.

En cas d'arrachement de la carte, il faut donc un mécanisme simple, implicite et instantané qui permet de défaire les instructions exécutées depuis le début du traitement de l'APDU. Les mécanismes simples de reprise sur panne utilisés jusqu'à maintenant deviennent impossibles pour ce niveau d'atomicité, car ils sont généralement basés sur des buffers d'instructions, dont la taille est trop limitée .

Ainsi, seule l'intégration dans les cartes, mêmes les plus simples, d'un mécanisme de reprise sur panne performant, de type de ceux utilisés dans les modèles transactionnels, permettra de traiter de manière efficace les problèmes d'arrachement de carte en cours d'application.

III.3.3 Atomicité Intra-Session :

Ce degré d'atomicité vise à rendre atomique l'exécution de plusieurs APDU, faisant toute partie de la même session, dans la carte. En effet, de manière à mieux gérer la reprise sur panne, le programmeur peut désirer rendre atomique l'exécution de plusieurs commandes APDU.

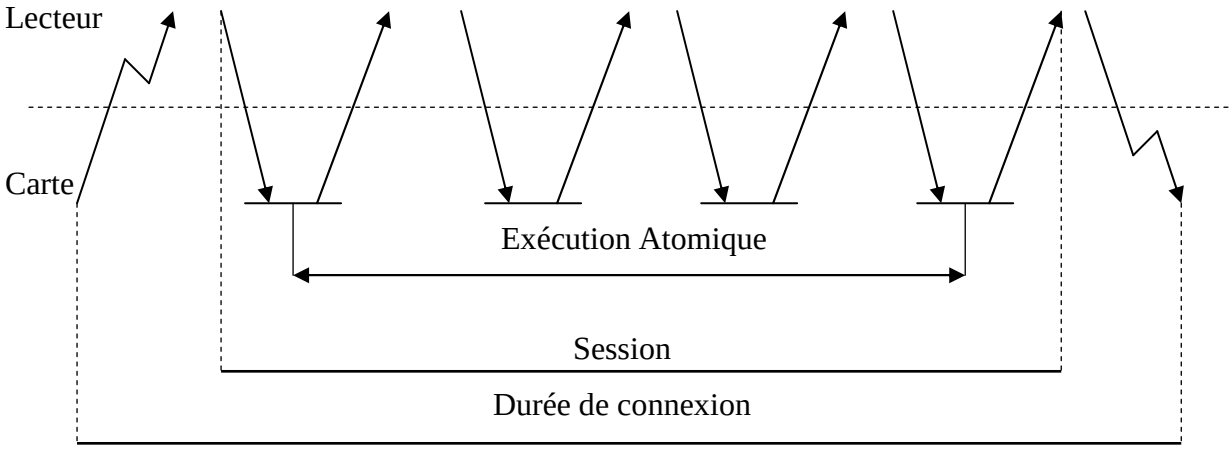


Figure III.bb : APDU atomiques intra-session [17]

Un exemple simple de traitement comprenant plusieurs APDU est le chargement d'un fichier ou d'un service dans la carte à microprocesseur. Si le chargement échoue, il faut défaire tous les traitements depuis la première APDU.

Un autre exemple est celui de l'invocation d'une méthode sur un objet dans la carte de type Java Card. Il faut invoquer la méthode, passer les paramètres et recevoir les résultats (dans le cas où il y a un retour). Cela se fait en plusieurs ordres APDU, et si la carte subit une panne, tout doit être défait (comme si la méthode n'avait pas été invoquée).

Contrairement au mécanisme qui rend l'exécution d'une APDU atomique, le mécanisme mis en jeu ici devra être explicite, pour plusieurs raisons :

- **Cohérence carte / terminal** : le terminal devra être prévenu que les traitements effectués par la carte à microprocesseur sont susceptibles d'être défaits,
- **Exécution d'une transaction** : il faut pouvoir délimiter les ordres que l'on veut atomiques. En effet, nous sommes dans le cas où l'on exécute une transaction dans la carte. Le « Begin_Transaction » et « Commit_Transaction » définis pour certaines cartes prennent donc ici tout leur sens.

Ce type d'exécution atomique représente bien ce qui est actuellement désiré par l'industrie carte, pour rendre la carte totalement résistante à l'arrachement. Avec ce type de carte, on ne peut exécuter qu'une seule transaction à la fois. La seule propriété à implanter est donc un mécanisme de reprise sur panne gérant une seule transaction simultanée.

Dans ce cas, que l'on choisisse un mécanisme à base de journalisation, ou à base de pages ombres, les traitements relatifs à la validation ou bien à l'abandon d'une transaction ne seront pas très lourds à gérer, car :

- une seule transaction implique un seul journal, ou une table d'indirection simple,
- tous les problèmes d'abandon en cascade ou de compensation de transaction sont inexistants.

Remarque : Dans le cas de ces applications simples, il n'est pas utile d'intégrer dans le système de la carte un mécanisme de contrôle de concurrence, seul un mécanisme de reprise sur panne suffit.

III.3.4 Transaction Inter-Session et Intra-Connexion :

La figure III.bc est la représentation au niveau de la communication carte/terminal d'une application mettant en jeu plusieurs services d'une même carte. Une telle application, de part ses caractéristiques, et de part les caractéristiques de la carte à microprocesseur, nécessitait une exécution suivant le modèle transactionnel.

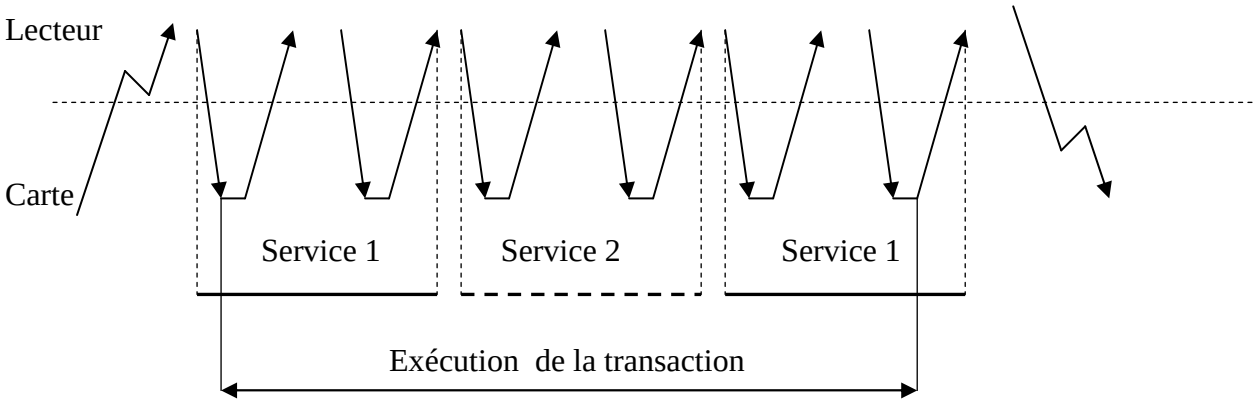


Figure III.bb : transactions sur plusieurs sessions [17]

Dans le cas de cette application, l'exécution selon le modèle transactionnel se poursuit sur plusieurs sessions (étant donné qu'un changement de session correspond à un changement d'exécution de service carte). Il faut donc respecter les propriétés ACID, non seulement sur plusieurs APDU, mais aussi sur plusieurs sessions.

L'interruption du service 1 pour traiter le service 2 par la carte à microprocesseur implique un certain nombre d'actions à effectuer par la carte :

- tout d'abord, la carte doit sauvegarder sur le support non volatile les données relatives au service 1, ainsi que le contexte d'exécution de ce service,
- ensuite il faut s'assurer que le service 2 n'a pas déjà commencé une exécution au préalable (auquel cas il faut recharger son contexte et reprendre l'exécution), et vérifier les droits d'accès à ce service.

Concernant la propriété d'isolation des transactions dans le cadre de l'exécution de la transaction selon la figure III.bb , nous avons deux possibilités :

- soit l'exécution du second service se fait dans le cadre de la même application que le service 1. Dans ce cas, le service 2 a le droit d'accès aux données, et de modifier les données déjà accédées par le service 1. Il n'y a donc pas de problème d'accès concurrents sur les données (l'identifiant de la transaction est le même).
- Soit l'exécution du service 2 n'a pas de rapport avec l'exécution du service 1 (il ne fait donc pas partie de la transaction). Dans ce cas là, les accès aux données doivent suivre la règle de sérialisation des transactions.

Pour la propriété de l'atomicité, nous devons, là aussi, différencier les deux cas :

- Si l'exécution du service 2 se fait dans le cadre de la même transaction que le service 1, une seule transaction sera considérée comme étant en cours dans la carte à microprocesseur. En cas d'annulation de cette transaction, il faudra *défaire* les actions de la transaction : les effets des services 1 et 2 seront globalement annulés.
- Si l'exécution du service 2 se fait en dehors de la transaction, alors nous nous retrouvons dans le cas d'une exécution simultanée de deux transactions.

Si la carte doit fonctionner comme dans le premier cas, le mécanisme de reprise sur panne sera semblable à celui utilisé pour garantir l'atomicité sur plusieurs APDU. En effet, cela revient à considérer qu'il n'y a plus qu'une seule transaction en cours dans la carte à microprocesseur.

A l'inverse, si l'on désire que la carte puisse accepter des requêtes ne concernant pas la transaction qu'elle est en train de traiter, alors le mécanisme de reprise sur panne doit être en mesure de traiter efficacement plusieurs contextes transactionnels simultanés.

Les cartes exécutants plusieurs services au cours d'une même connexion, doivent donc être munies des mêmes mécanismes que les serveurs « classiques ». Cependant, ces mécanismes devront être adaptés aux caractéristiques des cartes à microprocesseur.

Concernant les mécanismes de contrôle de concurrence, il est à noter que la carte contient peu de données, et que donc, en cas d'exécution concurrente de services, le risque d'accès concurrents à ces données est élevé. Il semble donc préférable d'utiliser un mode de contrôle de concurrence pessimiste.

Pour le mécanisme de reprise sur panne, les mécanismes connus, comme les pages ombres, ou la journalisation, devront être adaptés aux cartes à microprocesseur, pour diminuer leur complexité, ainsi que la taille mémoire utilisée.

III.3.5 Transaction Multi-Connexions :

Ce dernier cas de communication Carte/Terminal (voir figure III.bc) est la représentation de l'exécution d'une application sur plusieurs connexions de la carte à microprocesseur.

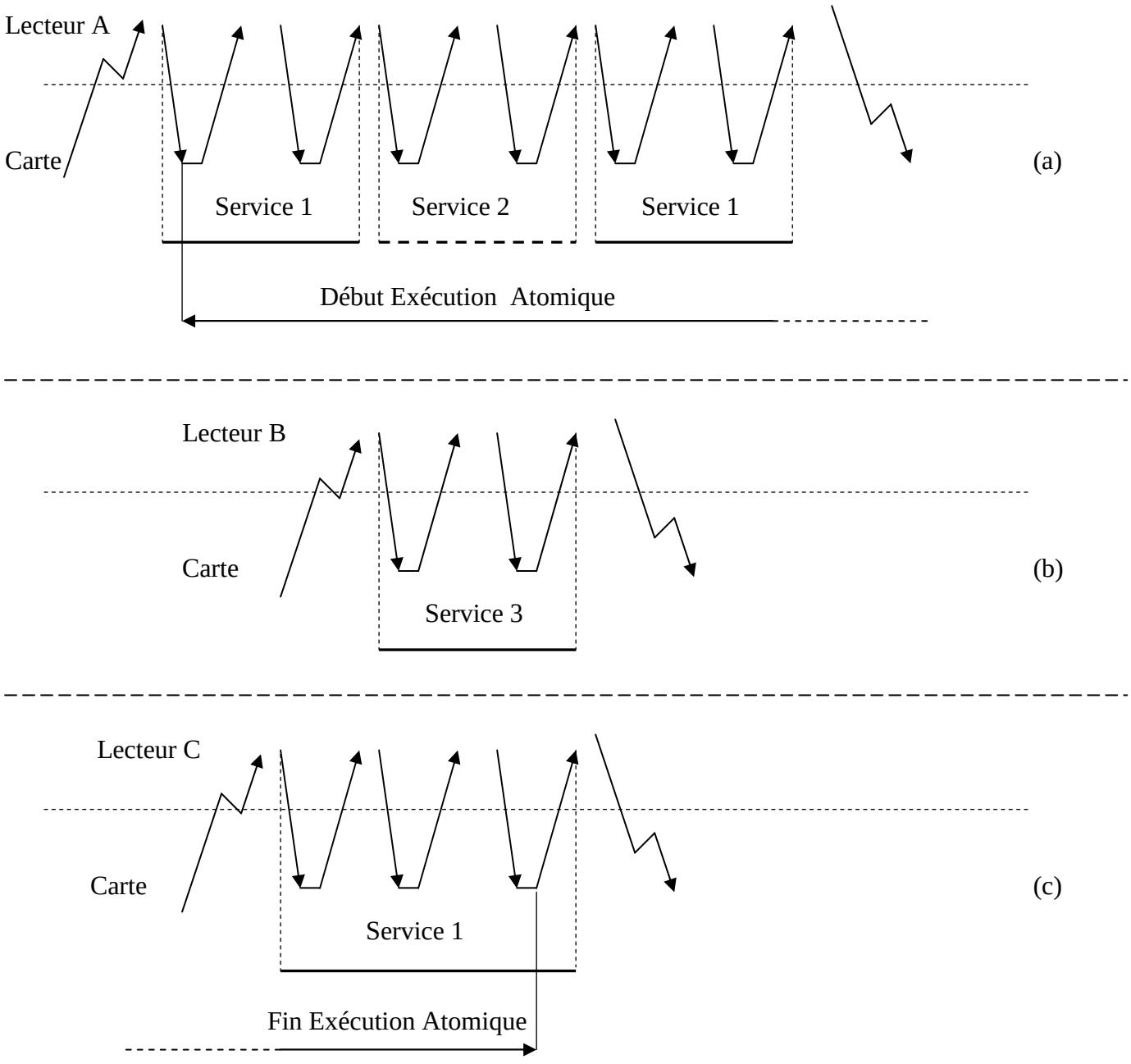


Figure III.bc : transactions sur plusieurs connexions [17]

Ce type de dialogue Carte / Terminal peut être divisé en trois phases :

- **la première phase** représente le début de l'application (c'est la phase (a) dans la figure III.bc). L'application est initialisée au niveau de la carte à microprocesseur. Cela signifie qu'un contexte transactionnel est créé, et que tous les services s'exécutant dans le cadre de cette transaction devront faire référence à ce contexte. Cette phase se produit une seule fois dans le cadre de l'application.
- **la seconde phase** concerne la reconnexion de la carte à microprocesseur après le début de l'application (c'est la phase (b) dans la figure III.bc). Cette phase peut bien entendu se produire plusieurs fois au cours de l'exécution de l'application. A une reconnexion de la carte à microprocesseur nous avons deux possibilités :
    - soit la carte est utilisée pour rendre un service sans aucun rapport avec l'application longue. Dans ce cas, il faut s'assurer que l'exécution de ce nouveau service n'entre pas en concurrence avec l'application longue. Cela peut passer par une exécution en *mode dégradé* (par mode dégradé on entend une exécution qui ne permet pas l'accès aux variables en cours d'utilisation) de la carte à microprocesseur. Cependant un tel fonctionnement peut être problématique en cas de prolongement de l'application longue (cela peut, par exemple bloquer l'utilisation du PME sur une longue période).
    - soit la carte se connecte pour poursuivre l'exécution de l'application longue. Dans ce cas là, le système de la carte à microprocesseur doit rétablir les données relatives à l'exécution de cette application, ainsi que les différents contextes.
- Enfin, **la troisième phase** concerne la dernière connexion de la carte relative à l'application longue. C'est dans cette phase que la décision de validation ou d'annulation de la transaction va être prise (c'est la phase (c) dans la figure III.bc).

Il est important de bien comprendre que la seconde phase peut constituer le départ d'une nouvelle application (soit simple, soit multiservices, soit longue). Cette carte devra donc être en mesure de gérer plusieurs contextes transactionnels.

La problématique du fonctionnement de la carte à microprocesseur entre deux connexions de l'application longue nécessite un contrôle de concurrence plus permissif que les contrôles de concurrence rigoureux, tels que le verrouillage à deux phases ou l'estampillage (il faut assouplir l'isolation).

De plus la carte étant déconnectée de son lecteur, le contenu de la mémoire volatile sera perdu. La fin d'une connexion de la carte à microprocesseur devra donc contenir une phase de sauvegarde du contexte de l'application, et une nouvelle connexion de la carte devra comporter une consultation de l'état de la carte (cette consultation pourrait être faite pendant l'ATR). Or, le protocole de connexion de la carte à microprocesseur ne comporte pas cette phase de chargement. La gestion de telles applications impose donc, en plus d'une gestion d'un modèle transactionnel avancé, une modification du modèle d'exécution des applications cartes, permettant d'effectuer des traitements dans la carte à la reconnexion de celle-ci, avant même qu'elle ne reçoive d'ordre du terminal.

III.4 Java Card et les transactions : [6], [24], [18], [w11]

Les cartes à puce sont un moyen émergeant et préféré dans les applications telles que le stockage de données confidentielles et la fourniture de service d'authentification dans un environnement mobile et distribué. Cependant avec les cartes à puce il y a un risque d'échec à tout moment durant l'exécution de l'applet. L'échec peut avoir lieu à une erreur de traitement, ou, plus souvent, un utilisateur de la carte à puce peut accidentellement enlever la carte du CAD, coupant le courant à la CPU de la carte et terminant l'exécution des applets. Le risque d'exécution incomplète présente un défi pour préserver l'intégrité des opérations sur des données dans la carte à puce.

Le JCRE fournit un mécanisme robuste pour assurer l'atomicité des opérations. Ce mécanisme est supporté en deux niveaux. Premièrement, la plateforme Java Card garantie que toute mise à jour à un champ simple dans un objet persistant ou un champ de classe simple est atomique. Deuxièmement, la plateforme Java Card supporte un modèle transactionnel, dans lequel une applet peut grouper un ensemble de mises à jour en une transaction. Dans ce modèle, l'atomicité de toutes les mises à jour est assurée [6].

III.4.1 Atomicité :

Sur la plateforme Java Card, l'atomicité veut dire que toute mise à jour à un champ d'objet persistant (même un élément de tableau) ou un champ de classe est garantie soit de terminer avec succès ou sinon restaurée à sa valeur originale si une erreur a lieu durant la mise à jour. Par exemple un champ d'un objet contient actuellement la valeur 1, et est en train d'être mis à jour avec la valeur 2. La carte est accidentellement arrachée du CAD au moment critique ou la carte écrit le champ. Quand le courant est rétabli, le champ n'est pas laissé avec une valeur aléatoire mais restauré à sa valeur précédente, 1.

Le concept de l'atomicité s'applique à la mémoire persistante. Il définit comment le JCRE manipule un élément de donnée simple dans le cas d'une perte de courant ou autre erreur durant la mise à jour de cet élément. La caractéristique d'atomicité du JCRE ne s'applique pas aux tableaux temporaires. La mise à jour d'un élément de tableau temporaire ne préserve pas la valeur précédente de l'élément en cas de perte de courant. Quand la carte est réinsérée dans le CAD, les éléments d'un tableau temporaire sont positionnés à leurs valeurs par défaut (`zéro`, `false`, ou `null`) [6].

III.4.2 Mises à jour en bloc dans un tableau :

La classe `javacard.framework.Util` fournit une méthode `arrayCopy` qui garantit l'atomicité des mises à jour en bloc de multiples données dans un tableau :

```
public static short arrayCopy
(byte[] src, short srcOff, byte[] dest, short desOff, short length)
```

La méthode `Util.arrayCopy` garantit soit que tous les octets sont correctement copiés ou que la tableau destinataire est restauré à ses valeurs d'octets précédentes. Si le tableau destinataire est temporaire,

la caractéristique de l'atomicité n'a pas lieu.

Cependant `arrayCopy` requière des écritures extra EEPROM pour supporter l'atomicité, et ainsi elle est lente. Une applet peut ne pas exiger l'atomicité pour les mises à jour de tableaux. La méthode `Util.arrayCopyNonAtomic` est fournie pour ce but :

```
public static short Util.arrayCopyNonAtomic
(byte[] src, short srcOff, byte[] dest, short desOff, short length)
```

La méthode `arrayCopyNonAtomic` n'utilise pas le moyen de la transaction durant l'opération de copie même si une transaction est en cours. Donc cette méthode sera utilisée uniquement si les contenus du tableau destinataire peut être laissé dans un état partiellement modifié dans l'événement de perte de courant au milieu d'une opération de copie. Une méthode similaire, `Util.arrayFillNonAtomic` remplit non atomiquement les éléments d'un tableau d'octets avec une valeur spécifiée :

```
public static short Util.arrayFillNonAtomic
(byte[] bArray, short bOff, short bLen, byte bValue)
```

III.4.3 Transactions :

L'atomicité garantit une modification atomique d'un simple élément de données. Cependant, un applet peut avoir besoin de mettre à jour atomiquement plusieurs champs différents dans plusieurs objets différents. Par exemple, une transaction de crédit ou de débit peut exiger d'une applet porte-monnaie d'incrémenter le numéro de transaction mettre à jour le solde porte-monnaie, et écrire un log de transaction , le tout comme une unité atomique de travail [6].

La technologie Java Card supporte un modèle transactionnel similaire à celui utilisé dans les bases de données, avec les procédures de validation ou d'annulation pour garantir que les opérations complexes peuvent être accomplies atomiquement ; soit elles se terminent avec succès ou soit leurs résultats partiaux ne prennent pas d'effet. Le mécanisme transactionnel protège contre les événements tels que perte de courant au milieu d'une transaction et contre des erreurs de programmation qui peuvent causer une corruption des données et empêcher toutes les étapes d'une transaction de terminer normalement.

III.4.3.1 débuter une transaction :

Pour entamer une transaction on utilise la méthode `beginTransaction` :

```
public static void beginTransaction()
```

Si une transaction est déjà en cours (transaction nesting depth level != 0), une exception `TransactionException` est lancée. Cette méthode peut ne rien faire si la méthode `Applet.register()` n'a pas encore été invoquée. Dans le cas d'un arrachement ou échec précédant un enregistrement à succès, l'environnement d'exécution Java Card annulera tout état persistent atomiquement mis à jour.

III.4.3.2 valider une transaction :

Une transaction est terminée en appelant la méthode `JCSystem.commitTransaction` :

```
public static void commitTransaction()
```

Le contenu du tampon de validation est validé atomiquement. Si une transaction n'est pas en cours (transaction nesting depth level == 0) alors une exception `TransactionException` est lancée.

L'appel de `beginTransaction` et `commitTransaction` peut se faire comme suit :

```
// débuter une transaction
JCSystem.beginTransaction();

// toutes les modifications dans un ensemble de mises à jour de données persistantes
// sont temporaires jusqu'à la validation de la transaction
...

// valider la transaction
JCSystem.commitTransaction();
```

Les changements de la transaction sont conditionnels – les champs ou éléments de tableau ont l'apparence d'être mis à jour. La lecture des champs éléments de tableau produira leurs dernières valeurs conditionnelles, mais les mises à jour ne sont validées qu'une fois la méthode `JCSystem.commitTransaction` appelée.

III.4.3.3 annuler une transaction :

Les transactions peuvent être annulées par une applet ou par le JCRE. Si une applet rencontre un problème interne, elle peut expressément annuler la transaction en appelant la méthode `JCSystem.abortTransaction`. L'annulation de la transaction provoque le JCRE à se débarrasser des changements faits durant la transaction et restaurer les champs ou éléments de tableau conditionnellement mis à jour à leurs valeurs précédentes. Une transaction doit être en cours quand la méthode `JCSystem.abortTransaction` est invoquée ; sinon le JCRE lance une '`TransactionException`'.

Quand le JCRE regagne le contrôle après un retour d'une applet avec une transaction en cours – et si l'applet n'a pas explicitement validé ou annuler la transaction en cours – le JCRE appelle automatiquement la méthode `abortTransaction`. Similairement, le JCRE annule la transaction si une exception est lancée dans la transaction et l'exception n'est pas traitée par l'applet.

Si le courant est coupé ou une erreur se produit durant une transaction, le JCRE invoque la procédure interne d'annulation lorsque la carte est remise en service pour restaurer les données impliquées dans la transaction à leurs valeurs pré transaction.

Dans tous les cas, les objets temporaires et persistants crées durant une transaction qui échoue (à cause de la perte de courant, réinitialisation de la carte, erreur de traitement, ou action d'annulation par programme) sont supprimés et leur mémoire libérée par le JCRE [6].

Remarque : En Java Card 3.0 la démarcation a changé : le status de transaction devient une propriété de la méthode, le champ de la transaction est le champ de la méthode.

Exemple : `@Transaction(REQUIRES_NEW)`

```
byte doSomething()
{
   byte[] ba ;
   shortField = 12 ;
   ba = new byte[10];
   shortArray = ba ;
   return ba[0];
}
```

Caractéristiques : - la transaction commence au début de la méthode,
- la transaction est validée avec un retour normal,
- la transaction est annulée par une exception,
- le comportement dépend des annotations.

III.4.3.4 Transaction emboîtée :

Contrairement à la plupart des transactions de bases de données, les transactions dans la plateforme Java Card 2.2.2 (ou antérieures) ne peuvent pas être emboîtées. Il ne peut y avoir qu'une transaction en cours en un temps donné. Cette exigence est due aux ressources limitées des cartes à puce.

Si `JCSystem.beginTransaction` est appelée alors qu'une transaction est en cours, le JCRE lance une '`TransactionException`'. Une applet peut découvrir si une transaction est en cours en invoquant la méthode `JCSystem.TransactionDepth`. La méthode retourne 1 si une transaction est en cours, 0 sinon.

Remarque : En Java Card 3.0, les transactions peuvent être emboîtées.

III.4.3.5 Capacité de validation :

Pour assurer l'annulation des transactions non validées, le JCRE maintient un **tampon de validation** (*commit buffer*) où le contenu original des champs mis à jour est stocké jusqu'à ce que la transaction est validée. Si un échec se produit avant que la transaction ne se termine, les champs concernés dans la transaction sont rétablis à leur contenu original à partir du tampon de validation. Plus il y a des opérations dans le bloc de transaction, plus large doit être le tampon de validation pour les prendre en considération.

La taille du tampon de validation varie d'une implémentation à une autre, suivant la mémoire carte disponible. En général, le tampon de validation alloué dans une implémentation JCRE est suffisamment large pour prendre en considération tous les besoins des applets – une applet accumule typiquement des dizaines d'octets durant une transaction. Cependant, à cause du fait que les ressources de la carte à puce sont limitées, il est important que seules les mises à jour dans une unité logique d'opérations sont inclues dans une transaction. Mettre beaucoup de choses dans une transaction peut ne pas être possible.

Avant d'aborder une transaction, une applet peut tester la taille du tampon de validation disponible face à la taille des données nécessitant une mise à jour atomique. La classe `JCSystem` fournit deux méthodes pour aider les applets à déterminer de combien est la capacité de validation disponible dans implémentation d'une plateforme Java Card.

- `JCSystem.getMaxCommitCapacity()`retourne le nombre total d'octets dans le tampon de validation.
- `JCSystem.getUnusedCommitCapacity()` retourne le nombre d'octets inutilisés dans le tampon de validation.

En plus de stocker les contenus des champs modifiés durant une transaction, le tampon de validation stocke des données additionnelles de données supplémentaires (*overhead data*), comme les emplacements des champs. La quantité de données supplémentaires dépend du nombre de champs qui sont modifiés et de l'implémentation du système de transaction. La capacité de validation retournée par les deux méthodes est le nombre total d'octets de données persistantes – incluant les supplémentaires – qui peuvent être modifiés durant une transaction.

Si la capacité de validation est excédée durant une transaction, le JCRE lance une '`TransactionException`'. Quand même, la transaction reste en cours à moins qu'elle soit explicitement annulée par l'applet ou par le JCRE.

III.4.3.6 TransactionException :

Le JCRE lance une '`TransactionException`' si certains types de problèmes, tels que l'emboîtement de transaction ou un débordement du tampon de validation sont détectés avec la transaction.

`TransactionException` est une sous classe de `RuntimeException`. Elle fournit un code raison pour indiquer la cause de l'exception. Ce qui suit est les codes raisons définis dans la classe `TransactionException` : (pour Java Card 2.2.2 ou antérieures)

- IN_PROGRESS – `beginTransaction` est appelée alors qu'il y a déjà une transaction en cours.
- NOT_IN_PROGRESS – `commitTransaction` ou `abortTransaction` a été appelée alors qu'il n'y a aucune transaction en cours.
- BUFFER_FULL – durant une transaction, une mise à jour à la mémoire persistante a été entamée et qui a provoqué le débordement du tampon de validation.
- INTERNAL_FAILURE – un problème interne fatal a eu lieu dans le système transactionnel.

Si une '`TransactionException`' n'est attrapée par l'applet, elle sera attrapée par le JCRE. Dans ce dernier cas, le JCRE annule automatiquement la transaction.

III.4.3.7 Variables locales et objets temporaires durant une transaction :

Il faut avoir conscience que seules les mises à jour aux objets persistants participent dans une transaction. Les mises à jour des objets temporaires et variables locales (et les paramètres des méthodes) ne sont jamais défaits indépendamment du fait que qu'ils soient ou non « à l'intérieur » de la transaction. Les variables locales sont créées sur la pile Java Card, qui réside en RAM.

Le fragment de code suivant illustre trois opérations de copie impliquant un tableau temporaire `key_buffer`. Quand la transaction est annulée, ni les opérations de copie ni la mise à jour simple de l'élément de `key_buffer` dans la boucle for ne sont protégés par la transaction. Similairement, la variable locale `a_local` retient la nouvelle valeur 1.

```
Byte[] key_buffer = JCSystm.makeTransientByteArray
                    (KEY_LENGTH , JCSystem.CLAR_ON_RESET);

JCSystem.beginTransaction();

Util.arrayCopy(src, src_off, key_buffer, 0, KEY_LNGTH);
Util.arrayCopyNonAtomic(src, src_off, key_buffer, 0, KEY_LNGTH);

for (byte i = 0; i < KEY_LENGTH; i++)
    key_buffer[i] = 0;

byte a_local =1;

JCSystem.abortTransaction();
```

Parce que les variables locales ou les éléments d'un tableau temporaire ne participent pas à une transaction, la création d'un objet et l'assignation de l'objet à une variable locale ou à un élément de tableau temporaire doit être considéré prudemment. Voici un exemple de code :

```
JCSystem.beginTransaction() ;

// ref_1 est un champ d'une instance (d'objet)
ref_1 = JCSystem.makeTransientObjectArray
        (LENGTH, JCSystem.CLEAR_ON_DESELECT);

// ref_2 est une variable locale
ref_2 = new SommeClass();

// tester le status
if (!condition)
   JCSystem.abortTransaction();
else
   JCSystem.commitTransaction();

return ref_2;
```

Dans l'exemple, le champ d'instance `ref_1` stocke une référence à un objet temporaire, et la variable locale `ref_2` stocke une référence à un objet persistant. Comme décrit précédemment, si la transaction est annulée les objets persistants et temporaires créés durant la transaction sont automatiquement détruits. Ceci n'a pas d'effet de bord sur le champ d'instance `ref_1`, parce que son contenu est remis à sa valeur originale si la transaction ne termine pas normalement. Cependant, un problème potentiel a lieu à la ligne suivante, quand un objet nouvellement créé est assigné à une variable locale. Suite à un échec de transaction le JCRE supprime l'objet ; cependant `ref_2` reste pointant l'emplacement où aucun objet n'existe plus. La situation devient pire si `ref_2` est utilisé ultérieurement comme une valeur de retour. Dans ce cas l'appelant reçoit un pointeur pendant.

Pour éviter de générer un pointeur pendant, qui compromet la sécurité du langage Java, le JCRE s'assure que les références aux objets créés durant une transaction annulée sont mis à `null`. Par exemple, si la méthode `abortTransaction` est invoquée, la variable locale `ref_2` est mise à `null`. Cette solution peut ne pas être idéale, mais évite la violation de sécurité.

Cet exemple n'est pas applicable à la plupart des applets, parce que la création d'objets dans une méthode est fortement découragée. Quand cela est possible, une applet doit allouer tous les objets dont elle a besoin durant l'initialisation de l'applet. Cependant, un implémenteur d'installeur Java Card peut avoir besoin de traiter avec une création d'objet considérable dans une transaction et devra éviter le scénario décrit dans le code [6].

III.4.4 Contrôle de concurrence :

Bien que la technologie Java Card assure l'atomicité des transactions, elle est dépourvue du contrôle de concurrence [18]. Or, le contrôle de concurrence peut s'avérer primordial dans les nouvelles applications pour résoudre le problème des accès concurrents et simultanés aux données partagées. Prenons l'exemple d'une carte qui regrouperait un Porte Monnaie Electronique et une Carte Bancaire. Lorsque le montant du Porte Monnaie Electronique est insuffisant pour effectuer un achat, celui-ci est crédité par une application de virement du compte bancaire. En Java Card 2, cela est difficile, sinon impossible, si les applications appartiennent à des packages différents ; c'est également le cas dans les diverses propositions de cartes multi-applicatives, les services sont cloisonnés, entre autres, pour des besoins de sécurité. Donc aucune coopération n'est prévue entre différents services installés sur la carte.

Cependant, la possibilité de partage de données communes est possible en Java Card depuis la version 2.2.2, mais il faudrait que les applets qui se partagent les données appartiennent au même package : une applet A peut accéder aux objets d'une applet B si A et B appartiennent au même package. En plus Java Card 2.2.2 a introduit la notion de canaux logiques où un terminal peut ouvrir plusieurs sessions avec la carte (1 à 20 sessions), et chaque session utilise un canal logique. Une applet peut être sélectionnée sur un seul canal logique de manière à avoir plusieurs applets sélectionnées en concurrence [w3].
Une même applet peut être sélectionnée simultanément sur plusieurs canaux logiques.

Cela étant, on peut appliquer une technique de contrôle de concurrence, comme celles décrites en III.2, avec la contrainte que la durée de vie maximale d'une transaction est limitée à la durée d'une communication APDU [24].

Pour implémenter notre protocole de « verrouillage multiple » (conçu et décrit au paragraphe III.2.4.4.5), on sera heurté au problème d'absence de primitives de synchronisation en Java Card 2 (ou antérieure). Les procédures `<bloquer une transaction>` et `<Réveiller une transaction>` nécessitent, en effet, d'être implémentées comme un `P` et un `V`, respectivement, sur des sémaphores.

Mais en Java Card 3.0, le problème cité précédemment n'a pas lieu car Java Card 3.0 propose du multithreading avec des mécanismes de synchronisation qui pourront répondre aux questions posées dans ce paragraphe [31], [w11].

Une autre solution consiste à intégrer au cœur du système d'exploitation le mécanisme de contrôle de concurrence [17].

III.5 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté les propriétés du modèle transactionnel (nommées propriétés ACID), ainsi que les mécanismes nécessaires à leur implantation. Ces mécanismes sont :

- Le mécanisme de contrôle de concurrence assure la propriété d'Isolation des actions entre transactions.
- Le mécanisme de reprise sur panne assure l'Atomicité et une partie de la durabilité.

Nous avons également présenté quelques normes et services transactionnels existants. Puis nous avons discuté des mécanismes transactionnels en Java Card.

Conclusion

L'utilisation des cartes à puce est en plein développement, grâce aux technologies telles que l'internet ou la téléphonie mobile. Ce développement nous pousse à mieux utiliser ce composant, à la fois synonyme de très haute sécurité, mais aussi de mobilité du code et des données qu'il contient.

En parallèle, la technologie mise en jeu dans les cartes à puce se rapproche de plus en plus des technologies utilisées dans les systèmes informatiques traditionnels (tant au niveau matériel que du logiciel embarqué). Ces avancées permettent des implantations d'applications de plus en plus élaborées au sein des cartes.

Cependant, ces nouvelles applications se heurtent à certaines caractéristiques qui sont problématiques pour les cartes à puce comme les risques de panne propres aux cartes, rendent difficiles le développement et la gestion de nouvelles applications mettant en jeu plusieurs partenaires.

Dans ce mémoire nous utilisons le modèle transactionnel four faciliter la tâche du programmeur dans les accès concurrents aux données des cartes et des pannes.

Dans le cadre du modèle transactionnel, nous avons conçu un protocole de gestion des transactions (plus exactement contrôle de concurrence) baptisé verrouillage multiple.

L'implantation d'un tel modèle dans la carte à puce passe par l'intégration, au cœur du système d'exploitation de cette carte, d'un mécanisme de reprise sur panne et d'un mécanisme de contrôle de concurrence. La carte ainsi obtenue est capable d'assurer les propriétés ACID des transactions lors de la modification des données qu'elle contient.

En guise de perspectives à ce travail, plusieurs extensions peuvent être envisagées :

- étudier l'équité du protocole de verrouillage multiple,
- étendre le protocole de verrouillage multiple aux systèmes distribués.
- proposer une implémentation du protocole du verrouillage multiple en Java Card 3.

Bibliographie

Bibliographie

[1] N.S. Barghouti , G.E. Kaiser : « Concurrency control in advance database applications », ACM Computing Survey, vol. 23, n° 3, Sept. 1991 ; pp. 270-317

[2] P.A. Bernstein , V. Hadzilacos , N. Goodman : « Concurrency control and recovery in database systems », Addison-Wesley, 1987

[3] Jérôme Besancenot , Michèle Cart , Jean Ferrié , Rachid Guerraoui , Philippe Pucheral , Bruno Traverson : « Les systèmes transactionnels , concepts, normes et produits » Hermes, 1997

[4] Serge Chaumette : « Sécurité et vérification d'applications embarquées en environnement Java Card » , LaBRI – Université de Bordeaux I, 2001

[5] L. Cheikh , B. Hocine : « Introduction à la technologie Java Card : Conception et réalisation d'une application porte-monnaie électronique » , mémoire d'ingénieur, Université M.MAMMERI de Tizi-Ouzou, 2007

[6] Zhiqun Chen : « Java Card Technology for smart cards : Architecture and Programmer's Guide » , Addison-Wesley, 2000

[7] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger : « The Notions of Consistency and Predicate Locks in a Database System » , Comm.ACM vol. 9 nr.11, 1976, pp. 624-633.

[8] Georges Gardarin : « Bases de données » , Eyrolles 2005

[9] G. Gardarin , P. Valduriez : « Relationnal Databases and Knowledge Bases » , Addison-Wesley Publishing Company , 1989

[10] J.N. Gray : « Notes on Database Operating Systems » , in Operating Systems – An advanced Course, R. Bayer Ed., Springer-Verlag, 1978

[11] J.N. Gray : « The Transaction Concept : Virtues and Limitations » , Proc. of the 7th Intl. Conf. on Very Large Databases, IEEE Ed., pp. 144-154, 1981

[12] Jim Gray , Andreas Reuter : « Transaction Processing – Concepts and Techniques » , 1999

[13] Gilles Grimaud : « Camille : un système d'exploitation ouvert pour carte à microprocesseur » , Thèse de doctorat , Université des sciences et technologie de Lille , 2000

[14] R.C. Holt : « Some Deadlock Properties of Computer Systems », ACM Computing Surveys, Vol. , n° 3, pp. 179-196, Sept 1972

[15] International Standard Organisation (ISO) : « Distributed Transaction Processing OSI-TP » Norme ISO/IEC 10026, 1996

[16] M. Kacel et Y. Laguel : « Vers l'implémentation de l'approche réactive dans la plateforme Java Card » , mémoire d'ingénieur, Université M.MAMMERI de Tizi-Ouzou, 2007

[17] Sylvain Lecomte : « COST-STIC : Cartes Orientées Services Transactionnels et Systèmes Transactionnels Intégrants des Cartes » , thèse de Docorat , Université des sciences et Technologie de Lille , 1998

[18] Sylvain Lecomte , Didier Donsez : « Intégration d'un Gestionnaire de Transactions dans les cartes à microprocesseur » , LIFL et LIMAV

[19] Pei-Jyun Leu and Bharat Bhargava : « Clarification of Two Phase Locking in Concurrent Processing » , IEEE Transactions on Software Engineering, vol. 14, no. 1 , January 1988 pp. 122-125

[20] B. Lampson , H. Sturgis : « Crash Recovery in Distributed Data Storage System », Xerox Technical Report, Palo Alto, Xerox Research Center, 1976

[21] R.A. Lorie : « Physical Integrity in a Large Segmented Database », ACM Transactions on Database Systems, vol. 2, n°1, pp. 91-104, Mars 1977

[22] C. Mohan, B. Lindsay, R. Obermark : « Transaction Management in the R* Distributed Database Management System », ACM Trans. on Database Syst., vol. 11, n°, Dec. 1986

[23] J.E. Murphy : « Ressource Allocation with Interlock Detection in a MultiTask system » Proc of AFIPS-FJCC Conf., vol. 33, n° 2, pp. 1169-1176, 1968

[24] Marcus Oestreicher : « Transactions in Java Card » , IBM Zurich Research Laboratory

[25] Cedric Pradel : « Specification, conception et prototypage d'une série de test de performance pour la technologie Java Card » , CNAM, 2006

[26] Wolfgang Rankl , Wolfgang Effing : « Smart Card Handbook » , John Wiley & Sons, Ltd, 2003

[27] M. Ruffin : « KITLOG Un service de journalisation générique » , thèse à l'université Paris VI, Septembre 1992

[28] A.S. Tanenbaum : « Systèmes d'exploitation » , Nouveaux Horizons , 2007

[29] Ch.Tavernier : « Les Cartes à puce – théorie et mise en œuvre » , Dunod , 2007

[30] J.S.M. Verhofstad : « Recovery Techniques for database systems », ACM Computing Surveys Vol. 10, n° 2, pp. 167-195, Juin 1978

[31] Eric Vétillard : « The art of Java Card 3.0 programming », Trusted Labs, December 2008

[32] W.E. Weihl : « Commutativity-based concurrency control for abstract data types » , IEEE Transactions on Computers, vol. 37, n° 12, 1988 ; pp. 1488-1505

[33] B.N.Bershad , S.Savage , P.Pardyak , E.G.Sirer , M.Finczynski , D.Beker , S.Eggers , C.Chambers : « Extensibility, Safety and performance in the spin operating system » In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267-284, Copper Mountain Resort, CO, USA, dec 1995

[34] D.Cheriton , K.Duda : « A caching model of operating system kernel functionnality » In *Proceedings of th First Sysmposium on Opearating Systems Design and Implementation*, pages 179-193, nov 1994

[35] Dawson R.Engler , M.Frans Kaashoek : « Exterminate all operating system abstractions » In *Poceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)* , pages 78-83 Orcas Island, Washington, May 1995

[36] H.Härtig , M.Hohmuth , J.Lidtke , S.Schönberg , J.Wolter : « The performance of micro-kernel based systmes » In *Poceedings of the sixteenth ACM Symposium on Operating Systems Principles*, 1997

[37] M.Seltzer , Y.Endo , C.Small and K.Smith : « Dealing with disaster : Surviving misbehaved kernel extensions ». In *Proceedings of the second Symposium on Operating Systems Design and implementation*, pages 213-228, 1996

Webliographie :

[w1] http://fr.wikipedia.org/wiki/Carte_à_puce

[w2] http://www.cnam.fr/elau/polycop/images/intro_carte_%E0_puce.pdf

[w3] http://cedric.cnam.fr/~bouzefra

[w4] http://deptinfo.cnam.fr/~paradinas/cours/ValC-IntroJavaCard.pdf

[w5] htp://www.irisa.fr/lande/denney/javacard_survol.ps

[w6] http://developers.sun.com/mobility/javacard/articles/javacard1/

[w7] http://developers.sun.com/mobility/javacard/articles/javacard2/

[w8] http://developers.sun.com/mobility/javacard/articles/javacard3/

[w9] http://rsa.esigetel.fr/Doc/Autres/Sar_ISI/Systemes%20transactionnels.pdf

[w10] http://www.cs.ru.nl/~woj/papers/download/esmart2006.pdf

[w11] http://java.sun.com/javacard/