# Middleware for Smart Cards

Chapter 16 of the book “Middleware for Communications”, John Wiley & Sons 2004

Harald Vogt*, Michael Rohs*, Roger Kilian-Kehr+

* Swiss Federal Institute of Technology (ETH Zurich)

+ SAP Corporate Research

## *Introduction*

Smart cards are credit card-sized plastic cards with an integrated microcontroller chip. This chip is protected against physical and logical tampering, thus unauthorized access to internal data structures is virtually impossible. This makes a smart card an excellent device for storing secret cryptographic keys and other sensitive data. In practice, smart cards are used for applications like digitally signing documents, ticketing, controlling access to desktop computers, and authenticating the users of mobile phone networks. Sometimes, smart card functionality is provided by other appliances such as USB “crypto tokens”, rings, or the GSM SIM (subscriber identity module). Although these devices look differently, they technically just differ in the interface to the host they connect to, thus we will treat them as smart cards as well.

In order to work together with a host computer, smart cards require an additional device that provides an electrical interface for data exchange, a so-called smart card reader. They nowadays can be found built into an increasing number of desktop computers. This makes smart card services available to the full range of PC and Web/Internet applications and lets them play a major role in payment schemes and for access control on the Internet.

The need for middleware and system support became immediately clear when smart cards started to gain ground on the common computing platforms. Application developers cannot be expected to deal with the large number of different, manufacturer-dependent interfaces that are offered by smart card readers and smart card services. Therefore, smart card middleware is necessary for mediating between these services and application software. In such a system, it should be possible to replace one type of smart card with another one offering similar features, without affecting the application level. The major requirements to smart card middleware are therefore encapsulation of communication specifics, interoperability, and system integration.

Standardization plays a major role in the smart card business. The lower system and communication layers have been standardized for a long time. The ISO 7816 standard [1] was established around 1989 and specifies not only the form factor and the electrical properties of the communication interface, but also a basic file system and card holder verification (the well-known PIN), among other features. Ever since, it was extended to embrace new requirements of smart card vendors. For the integration with host computers, PC/SC [2] is the de-facto industry standard since it was adopted by the major desktop operating systems.

In this chapter, we present an overview of the existing approaches to smart card middleware. Some of them were inspired by the development of powerful microprocessors for smart cards that made it possible to run a Java interpreter. Generally, the availability of the Java programming language for smart cards increased their popularity and made lots of new applications possible. As we will see later in the chapter, this also made it easier to integrate them into distributed computing environments. This shows that, with the right kind of middleware, the power of smart cards can be effectively utilized.

# *ISO 7816*

We start by briefly introducing the most basic communication principles of ISO-compliant smart cards. The electronic signals and transmission protocols of smart cards are standardized in ISO 7816 [1]. This standard consists of several parts, which define the characteristics of smart cards on various layers. It comprises physical characteristics, like the position of the contacts on the card, electrical characteristics, transmission protocols between card and card reader, instruction sets of frequently used card commands, and more. ISO 7816 does not cover the interface between the card reader and the PC.

## Communication between Card and Card Reader

When a card is inserted into the card reader, the card is powered up and the RST (reset) line is activated. The card responds with an "answer to reset" (ATR), which gives basic information about the card type, its electrical characteristics, communication conventions, and its manufacturer. An interesting portion of the ATR are the "historical characters" or "historicals". The historicals are not prescribed by any standard. They mostly contain information about the smart card, its operating system, version number, etc. in a proprietary format. In addition to the ATR, ISO 7816-4 [3] defines an ATR file that contains further information about the ATR. The ATR and the ATR file play an important role for the integration of smart cards into middleware systems. The actual communication between card and card reader takes place according to the protocol parameters specified in the ATR.

### Communication Protocols

The ATR contains information about the communication protocols the card supports. The protocols used most often are called T=0 and T=1. T=0 is an asynchronous character-oriented half-duplex protocol that is relatively easy to implement. T=1 is a block-oriented half-duplex protocol. It is more complex than T=0 and ensures the reliable delivery of data blocks.

### Application Layer

On the application layer, requests and responses between card and card reader are exchanged in so-called application protocol data units (APDUs). Command APDUs are requests from the card reader to the card, while response APDUs contain the corresponding answers of the smart card. APDUs are defined independently of the communication protocol (T=0 or T=1).

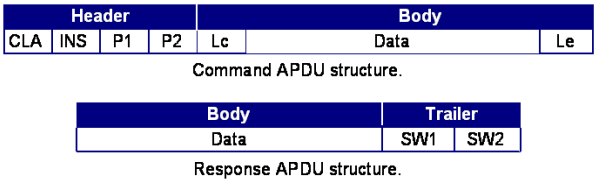


**Fig.1** Basic structure of command and response APDUs

**Command APDUs:** A command APDU specifies a request to the card. It consists of a header of fixed length and an optional body of variable length. The structure of command and response APDUs is shown in Fig. 1. The CLA byte specifies the instruction class, which also defines the requested card application (e.g., 0xA0 for GSM). The INS byte is the operation code of the instruction to execute. P1 and P2 are parameters for the operation. The body of a command APDU has a variable structure, depending whether data bytes are sent in the command APDU, expected in the response APDU, or both. The Lc byte specifies the number of data bytes, the Le byte the number of bytes expected in the response APDU.

**Response APDUs:** They consist of an optional body, containing the application data of the answer corresponding to the request, and a mandatory trailer, consisting of a status word that indicates the correct execution of the command or the error reason.

## Data Structures on Smart Cards

ISO 7816-4 specifies a file system as one possibility to organize the data on the card's EEPROM. All mutable data is stored in files of various types. Access to sensitive files can be bound to security conditions, like the authentication of the user to the card. Smart card files have an inner structure: they can be byte- or record-oriented. In addition to the core file system itself, ISO 7816-4 defines an instruction set to access and administer the files in a well-defined manner. The ISO 7816-4 file system is organized hierarchically: There are directory files containing other files, and elementary files, containing application data. Files consist of a header for administrative data – like file type, file size, and access conditions – as well as a body for the actual data. On many cards, the file hierarchy is fixed. In this case, files cannot be created or deleted, but only the contents of existing files can be changed.

Smart card files are named by 2 byte *file identifiers (FIDs)* and addressed by path names consisting of the concatenation of several FIDs. Before a file can be accessed, it has to be selected using a special SELECT FILE command APDU. Most file operations implicitly operate on the currently selected file.

*Dedicated files (DFs)* are directory files that group files of the same application or category. In addition to FIDs, DFs are addressed via *application identifiers (AIDs)*, which are 5 to 16 bytes long. An AID is used to select the card application that is contained in the identified DF. JavaCard applets, which are described below, are also selected using AIDs.

## Command Sets

ISO 7816-4 defines an extensive set of so-called "basic interindustry commands". These commands are not targeted to specific applications, but are generally applicable to a wide array of different needs, which is in contrast to the application-specific commands that are defined, e.g., for GSM SIMs [4]. The goal of specifying these commands is to unify access to smart card services. Unfortunately this goal has not been fully achieved, because some smart cards do not completely implement all commands or command options, while others provide vendor specific commands for the same purpose. It could also be argued that it is not useful to attempt a standardization of this kind at this relatively low level of abstraction. The basic interindustry commands can be classified as follows:

- **File selection commands:**
  SELECT FILE
- **Read and write commands (transparent EFs):**
  READ / WRITE / UPDATE / ERASE BINARY
- **Read and write commands (record-oriented EFs):**
  READ / WRITE / UPDATE / APPEND RECORD
- **Authentication commands:**
  VERIFY, INTERNAL / EXTERNAL AUTHENTICATE, GET CHALLENGE
- **Transmission-oriented commands:**
  GET RESPONSE, ENVELOPE, MANAGE CHANNEL

# *JavaCards*

A JavaCard [5] basically is a smart card that runs Java programs. It contains the JavaCard Runtime Environment (JCRE) [6] that is capable to execute a restricted version of Java byte code. JCRE is standardized and hides the specifics of the smart card hardware. Binary compatibility was a major motivation for developing JavaCard: "The standards that define the Java platform allow for binary portability of Java programs across all Java platform implementations. This 'write once, run anywhere' quality of Java programs is perhaps the most significant feature of the platform." [7]

Java is a secure and robust language, since it is type safe, it disallows direct memory access, and its execution is controlled by an interpreter. It is known to many software developers, which further lowers the hurdle of smart card programming. It has to be said though, that the significant resource constraints on smart cards fundamentally change the "feel" of programming a smart card compared to standard Java

programming on a PC. Programming a smart card in Java is, however, less error prone and more efficient than programming it in the assembly language of the smart card's microcontroller. Other examples of high-level languages for smart cards are Windows for Smart cards [8], MultOS [9], and BasicCard [10].

## Hardware Architecture

The JCRE, which includes the JavaCard virtual machine (VM) and a basic class library, is located in the ROM of the card. It takes on the role of the smart card operating system, controlling access to card resources, like memory and I/O. JavaCard applets and their persistent state are stored in the EEPROM. The RAM contains the runtime stack, the I/O buffer, and transient objects. To the external world, a JavaCard looks like an ordinary ISO 7816 compliant smart card, i.e. it communicates by exchanging APDUs. This means that JavaCards work in situations where legacy ISO 7816 cards have been used before. The downside of this approach is that JavaCard developers have to deal with APDUs. For each application, the developer needs to specify an APDU protocol, and manually encode and decode APDUs.

JavaCard version 2.2 has introduced a middleware layer – called JavaCard RMI – which hides the APDU layer and therefore relieves the developer from manually developing an APDU protocol. Java Card RMI will be described further down.

## Runtime Environment

Fig. 2 shows the JavaCard software architecture. The lowest layer implements the communication protocols, like T=0 and T=1, memory management, and cryptographic functionality. For performance reasons, this layer is partly realized in hardware and coded in the machine language of the smart card microprocessor. The central component of the architecture is the JavaCard virtual machine (VM), which provides an abstraction of the concrete hardware details, like the microprocessor instruction set, and controls access to hardware components, like I/O. It offers a runtime environment for JavaCard applets and isolates the individual applets from each other. The JavaCard API component provides access to the services of the JavaCard framework.

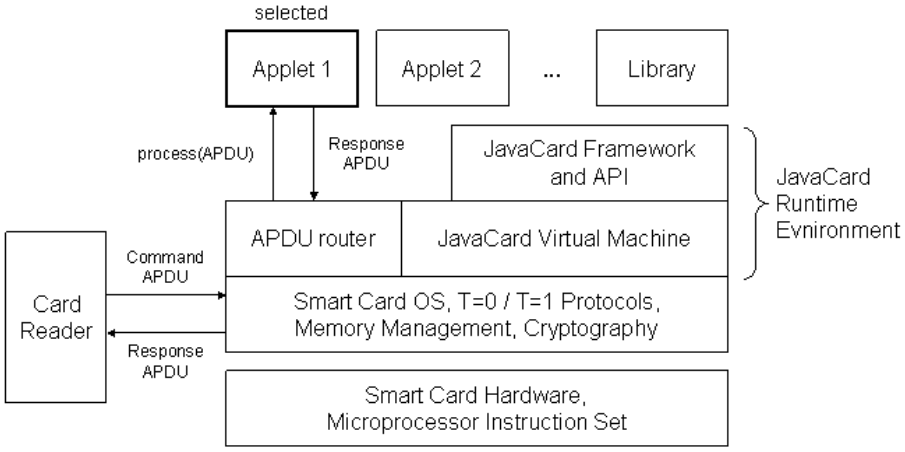


**Fig. 2** JavaCard software architecture

The actual functionality of the card is implemented as a set of JavaCard applets, each representing a different card service. In addition to applets, user libraries can be installed, which may be used by applets, but cannot be externally invoked.

The applet installer is an optional component of the JCRE, that – if it exists – accepts APDUs containing authorization data, the code of the applet to install, and linkage information. The JCRE specification describes applet installation and deletion in general terms only, in order to leave freedom of implementation details to smart card manufacturers. This led to various proprietary installer implementations, which is in contrast to the idea of a standardized open runtime environment. One example architecture for secure installation of new applications on multi application cards is the Global Platform

[11]. It comprises mechanisms for loading and installing application code and for general card content management.

A complete implementation of the Java 2 VM [12] is not feasible with current hardware, in particular if one considers that smart cards are low-cost mass products that are produced as cheaply as possible. To meet the constraints of smart card hardware, two strategies have been followed. Firstly the VM itself was thinned out and secondly the binary format of Java class files was simplified. Costly elements of the Java language like dynamic class loading, the standard Java security architecture, garbage collection, and object cloning have been removed. The data types `long`, `char`, `float`, and `double` are not available, and support for `int` is optional. The 16-bit `short` is the standard data type for integer operations. JavaCard does not use the – relatively verbose and flexible – class file format, but the CAP file format, which contains the binary data of a whole package. Linkage information is outsourced to a separate EXP file.

Conceptually, the JavaCard runtime environment acts as a server: It continuously runs in an infinite loop waiting for command APDUs, routing them to the currently selected applet for processing, and sending back response APDUs. An applet is selected for processing by using a special SELECT FILE APDU that contains its application identifier (AID)[1]. For each APDU the JCRE receives, it first checks whether it is a SELECT FILE APDU. If so, the current applet is deselected and the new one is selected. Subsequently, each APDU is dispatched to the selected applet by calling its "process" method. The "process" method takes the command APDU as a parameter.

Each applet runs in its own context. JCRE itself runs in a special context with extended privileges. The border between two contexts is called a "software firewall". Objects are associated with their originating context and can only be accessed from within this context, with one exception: To provide a way for applets to interact, an object of another context can be accessed if it implements the *javacard.framework.Shareable* interface.

JCRE provides a simple transaction processing system. It guarantees atomic changes to a related set of persistent objects that are included in a transaction. When the transaction successfully commits, all of its operations have been executed. When it fails, intermediate states are rolled-back to the state before the transaction. This is crucial for reliability and fault tolerance of security-critical operations. Changing single fields is atomic, as well as copying arrays with *Util.arrayCopy*. A set of methods controls the beginning, commit, and abort of a transaction.

The JavaCard 2.2 API is subdivided into seven packages:

- java.lang and java.io — JavaCard versions of some Java exceptions
- java.rmi — JavaCard version of the Remote interface
- javacard.framework — Base classes of the JavaCard framework
- javacard.framework.service — A framework for implementing an applet as a set of services
- javacard.security — Classes related to cryptography
- javacardx.crypto — Classes related to export-restricted cryptography

The central classes of the API are *Applet*, *APDU*, and *JCSystem*:

JavaCard applets are extensions of *javacard.framework.Applet*. They must implement the *process* method, which is called by the JCRE to handle incoming command APDUs, and the *install* factory method to create the applet object instance. Calling *register* completes the successful instantiation of a new applet. The example in the next section illustrates how these methods are used.

The class *javacard.framework.APDU* encapsulates communication with the external world according to the ISO 7816-3 standard. It manages the I/O buffer and sets the data direction of the half duplex communication. It works with the protocols T=0 and T=1. As we will see later, JavaCard RMI hides APDU handling from the implementer of card applications.

[1] It also contains a channel identifier, which is not discussed here.

*javacard.framework.JCSystem* is used for accessing various resources of the card and services of the JCRE. It provides methods for controlling transactions, for interactions between applets, and for managing object persistence.

## Developing JavaCard Applets

The JavaCard development kit is available for download from Sun's web site [13]. It includes a set of tools, a reference implementation for simulating smart cards, and documentation. A JavaCard applet is first compiled using a standard Java compiler, such as *javac*. The applet classes may only reference classes of the JavaCard API as well as any additional classes that are available on the card. The result is then converted into CAP and EXP files, containing the data for the applet package and linkage information, respectively. To perform the conversion process, the converter needs linkage information for the JCRE. To test the applet, it can be downloaded into a smart card simulator. The simulated card of the JavaCard reference implementation contains an installation applet that downloads the CAP file data in a series of APDUs.

In the following we give a step-by-step example to illustrate how to develop an applet with JavaCard. The example applet is a prepaid card with an initial balance of 1000 money units. There are operations for debiting a certain amount and for querying the remaining balance. If the amount to be debited is negative or greater then the remaining amount, an exception is thrown.

The first step in developing a JavaCard applet is to define an APDU protocol. The set of command APDUs, expected response APDUs, and error conditions have to be specified. The APDU protocol is the externally visible interface of the smart card and therefore has to be described precisely. It should include the preconditions for applying each command, the coding of the APDUs and the individual parameters, the possible errors and their associated status words. In our example, the APDU protocol looks as follows:

GET_BALANCE command APDU:

```
CLA  INS  P1   P2   Le
0x80 0x10 0x00 0x00 0x02
```

GET_BALANCE response APDU:

```
BaHi BaLo SW1  SW2
0xXX 0xXX 0x90 0x00
```

We assume signed two-byte monetary values. The first byte of the response contains the high-order byte of the balance, the second on contains the low-order byte. The status word 0x9000 (SW1, SW2) usually signals the successful execution of an operation.

DEBIT command APDU:

```
CLA  INS  P1   P2   Lc   AmHi AmLo Pin1 Pin2 Pin3 Pin4 Le
0x80 0x20 0x00 0x00 0x06 0xXX 0xXX 0xYY 0xYY 0xYY 0xYY 0x00
```

The data part of the command APDU contains the amount to be debited, high-order byte first, followed by a four-byte pin.

DEBIT response APDU (success):

```
SW1  SW2
0x90 0x00
```

DEBIT response APDU (bad argument):

```
SW1  SW2
0x60 0x00
```

DEBIT response APDU (PIN error):

```
SW1  SW2
0x60 0x01
```

After having specified the APDU protocol, the class structure and the individual methods of the applet classes need to be designed. The central component is an applet class (called *PrePaidApplet* in the example), which has to be derived from *javacard.framework.Applet*. Its *install* method needs to create an instance of the applet and register it with the JCRE. To work on incoming requests, the JCRE calls the

applet's *process* method. It checks class and instruction bytes of the command APDUs and dispatches them to the proper method. Finally, AIDs need to be assigned to the applet and its package.

```
package prepaid;

import javacard.framework.ISO7816;
import javacard.framework.APDU;
import javacard.framework.ISOException;
import javacard.framework.OwnerPIN;
import javacard.framework.JCSystem;

public class PrePaidApplet extends javacard.framework.Applet {

    // code of CLA byte in the command APDU header
    public final static byte CLA              = (byte)0x80;

    // codes of INS byte in the command APDU header
    public final static byte GET_BALANCE      = (byte) 0x10;
    public final static byte DEBIT            = (byte) 0x20;

    // status words
    public final static short SW_BAD_ARGUMENT = (short)0x6000;
    public final static short SW_PIN_ERROR    = (short)0x6001;

    // state
    private short balance;
    private OwnerPIN pin;

    private final static byte[] pinData =
        { (byte)0x31, (byte)0x32, (byte)0x33, (byte)0x34 };

    public PrePaidApplet() {
        balance = (short)1000; // initial balance
        pin = new OwnerPIN((byte)3, (byte)4);
        pin.update(pinData, (short)0, (byte)4);
        register(); // register the applet with JCRE
    }

    public static void install(byte[] aid, short s, byte b) {
        new PrePaidApplet();
    }

    public void process(APDU apdu) throws ISOException {
        if (selectingApplet()) return; // return if this is the selection APDU

        byte[] buffer = apdu.getBuffer();

        if (buffer[ISO7816.OFFSET_CLA] != CLA)
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

        switch (buffer[ISO7816.OFFSET_INS]) {
            case DEBIT:
                debit(apdu);
                return;
            case GET_BALANCE:
                getBalance(apdu);
                return;
            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
```

```
    public void debit(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        byte lc = buffer[ISO7816.OFFSET_LC];
        byte bytesRead = (byte)apdu.setIncomingAndReceive();

        if (lc != 6 || bytesRead != 6)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        if (!pin.check(buffer, (short)(ISO7816.OFFSET_CDATA + 2),
                               (byte)pinData.length))
            ISOException.throwIt(SW_PIN_ERROR);

        short amount = (short)
            ((buffer[ISO7816.OFFSET_CDATA] << 8) |
             (buffer[ISO7816.OFFSET_CDATA + 1] & 0xff));

        if (amount < 0 || amount > balance) {
            ISOException.throwIt(SW_BAD_ARGUMENT);
        }

        JCSystem.beginTransaction();
        balance -= amount;
        JCSystem.commitTransaction();
    }

    public void getBalance(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        short le = apdu.setOutgoing();
        if (le < 2) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

        apdu.setOutgoingLength((short)2);

        buffer[0] = (byte)(balance >> 8);
        buffer[1] = (byte)balance;

        apdu.sendBytes((short)0, (short)2);
    }

}
```

The *debit* method in the example shows how transactions can be used in JavaCard applets. In this case the transaction is not strictly necessary, because the JCRE guarantees the atomic update of an individual field, but has been included for didactic reasons.

This simple example shows that Java programming on a smart card fundamentally differs from programming Java on a full-fledged PC. The severe resource constraints have to be considered all the time. The lack of garbage collection means that all objects should be created at the time of applet construction and never dynamically during the processing of a request, except for transient arrays. JavaCard 2.2 allows for deleting objects upon request, but this feature should be used sparingly, because it is resource intensive. Writing to EEPROM, which is used for persistent storage on the card, is orders of magnitudes slower than writing to RAM. Even worse, an EEPROM cell can be overwritten only a limited number of times. Therefore, the number of EEPROM writing operations should be minimized.

## PC/SC: Data Communications

In 1997, an industry consortium formed by smart card manufacturers and IT companies released the specification of a standard smart card interface for PC environments, called PC/SC [2]. The main components of PC/SC have been integrated into the Windows operating system, and most card readers come with device drivers that are compatible with PC/SC. The specification was also taken up by an open source project for implementation within the Linux operating system, called MUSCLE [14].

PC/SC addresses the following issues:

- The lack of a standardized interface to card readers, which has significantly hampered the deployment of applications in large numbers.
- The lack of a high-level programming interface to IC cards.
- It was anticipated that IC cards would be shared by multiple applications in the near future, thus a sharing mechanism was required.
- To be widely deployed, such a framework must work with (nearly) all existing smart card technologies while being extensible to embrace technological advancements.

The PC/SC specification describes an architecture for the integration of smart card readers and smart cards with applications. The main concepts of this architecture are the Resource Manager (RM) and the Service Provider (SP). The RM keeps track of installed card reader devices and controls access to them. There is a single instance of the RM within a given system, and all accesses to smart card resources must pass through it. Thus, it can grant or deny applications access to smart card services, depending on access rights and concurrent accesses.

An SP represents a specific smart card type, providing a high-level interface to the functionality of that card type, e.g. file system operations or cryptographic operations. In principle, an SP can represent more than one card type, and one card type can provide more than one SP. However, for each card type, a "primary" service provider is identified. A database (which is part of the RM) keeps track of the mapping from card types (identified by their ATR) to SPs.

The RM is essentially part of the operating system. Its state is visible only through its interface functions, which are prefixed by *"SCard..."*. Before smart card resources can be addressed, an application has to establish a "context" with the RM through the *SCardEstablishContext* call:

```
SCARDCONTEXT hCardContext;
SCardEstablishContext(SCARD_SCOPE_USER, NULL, NULL, &hCardContext);
```

The resulting context handle is used in subsequent interactions with the RM. For example, when a new smart card type is introduced to the system. This step associates a card type with a service provider. It is necessary if the card type is to be recognized by the RM. After a card type is made known to the RM, the RM can notify applications when a smart card of that type is inserted into a card reader.

A card type is registered with the smart card database through the system call *SCardIntroduceCardType*. A type is identified by an ATR string (as described above) and an associated byte mask that rules out insignificant and volatile parts of an ATR. Additionally, a user-recognizable name can be assigned to a card type, which can be used later when searching for inserted smart cards. A card type needs to be registered only once. Later, when any card with a matching ATR is inserted, applications can query the database for supported interfaces. This mechanism decouples applications from specific smart cards: from the application's point of view, it is important to know whether a smart card can be used through a certain interface, for example providing user authentication. Details such as the command structure or the contents of the card's file system should be hidden from the application.

Note that there is a rather static model of smart card applications behind this architecture. It assumes that the functionality of a smart card does not change frequently, if at all. The mapping from smart card types to interfaces is fixed, and it is not possible for cards to advertise their capabilities dynamically. Thus, smart cards of the same type but of different configurations, such as Java cards with slightly different card applets, would have to be assigned different types. This requires that they respond with different ATRs and are introduced to the system separately, which might be hard to achieve in practice.

For the implementation of a service provider, it is necessary to be able to directly exchange APDUs with a smart card. The Resource Manager provides the *SCardTransmit* system call for this purpose. This function requires detailed knowledge of the command structure of the used smart card. It can also be used by applications, but then most of the interoperability capabilities of a PC/SC environment are lost. Within the MUSCLE implementation of PC/SC, which is still under development, this is currently the only means of interaction with smart cards, since service providers are not yet supported.

Some operations require the exchange of several APDUs that may not be interleaved by APDUs from other applications. For example, decrypting a large message requires that several APDUs, each containing a small data chunk, are sent to the card. An application that is not authorized by the user must not be allowed to intersperse its own data and thus gain access to confidential information. The RM grants temporary exclusive access for commands that are enclosed within a SCardBeginTransaction / SCardEndTransaction pair. The SCardEndTransaction call even allows resetting the card, which makes it necessary that the user authenticates herself again for subsequent security-sensitive operations.

In summary, a PC/SC compliant environment regulates access to smart card resources (card readers and smart cards) and provides applications with high-level interfaces to services offered by smart cards. A PC/SC implementation therefore acts as a middleware component for smart card applications. It supports the division of responsibility for the implementation of smart card-aware applications among three groups:

- the card reader manufacturers provide compliant drivers for their devices;
- smart card vendors implement service providers and registration routines for their cards;
- application developers make use of basic Resource Manager functionality but mainly rely on the service providers.

Information about the use of smart cards in the Windows environment can be found in the MSDN library [15] in the Security/Authentication section.

## *OpenCard Framework*

The OpenCard Framework (OCF) is a Java-based middleware for smart cards. OCF serves as an application platform, unifying access to a large variety of cards and terminals. In addition, it provides a standardized programming interface for application developers and developers of smart card services. In contrast to JavaCard, OCF focuses on the card-external part of smart card development.

OCF is realized as a set of *"jar"* files and has to be installed on the PC to which the terminal is connected. OCF runs in the VM of the application that uses OCF instead of as a separate OS service.

### Architectural Concepts

OCF consists of two subsystems: one for card terminals and one for card services. Card terminals encapsulate physical card readers. Card services are used by applications to talk to the actual services located on the card. The architecture of OCF is depicted in Fig. 3.

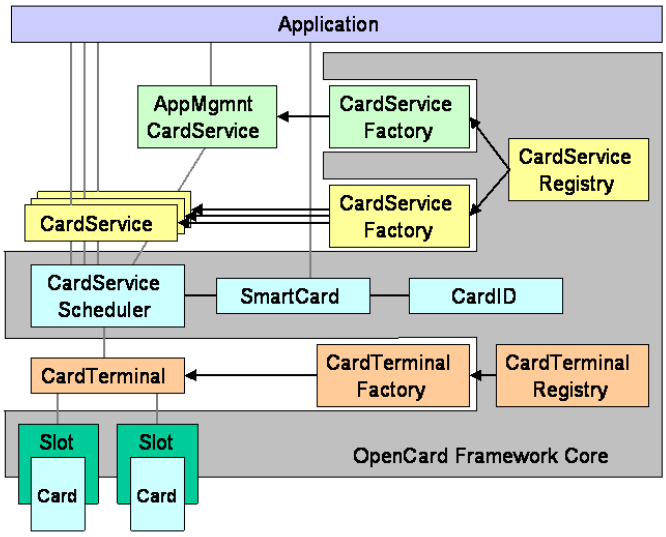


**Fig. 3** Architecture of the OpenCard Framework

**Card Terminals**

The card terminal layer abstracts from the reader device actually employed, thus guaranteeing the independence of an application from a particular reader type. Card reader manufacturers who want to make their devices available for OCF need to provide OCF drivers, which are realized as implementations of the abstract classes *CardTerminal* and *CardTerminalFactory*.

Each card reader is represented by a *CardTerminal* instance. Its card slots are modeled as instances of class *Slot*. The ATRs of a card are encapsulated in *CardID* objects. The card terminals connected to a PC are usually statically configured, but can also be added dynamically.

Mutually exclusive access to a card in a card slot is guarded by *SlotChannel* objects. An object owning a *SlotChannel* instance has exclusive access to a smart card, until the slot channel is released again. The class *CardTerminal* provides methods for checking for card presence, getting slot channel objects, and for sending and receiving APDUs. For extended functionality of special card readers, like number pads and finger print readers, OCF provides separate Java interfaces that can be implemented by the concrete *CardTerminal* implementations.

The singleton object *CardTerminalRegistry* records all terminals installed in the system. During startup, the registry reads configuration data from one or more "opencard.properties" files. At runtime, additional terminals might be added. The registry creates *CardTerminal* objects by using *CardTerminalFactory* objects, whose class names are listed in the configuration file. These factories are responsible for instantiating the OCF drivers for a particular family of terminals, for example those of a certain manufacturer. Upon card insertion and removal card terminals generate events. Objects can register at the registry for those events.

**Card Services**

The card service layer defines mechanisms for accessing card-resident functionality. It provides high-level interfaces to the individual card services and abstracts from particularities of the card operating system. In this layer, card services are represented as extensions of the abstract *CardService* base class. The *CardService* abstraction provides the means to integrate smart card functionality into Java applications and defines the application developer's view of a smart card service. For application developers, card services are the primary API, while the other components of OCF are less important. Developers of card services need a deeper understanding of the structure of the OCF, of course.

OCF defines some standard card services, like *FileAccessCardService* and *SignatureCardService*. The *FileAccessCardService* interface encapsulates smart cards that adhere to the ISO 7816-4 standard. It provides access to binary and structured files, as well as to the information stored in the file headers. Around these low-level interfaces, which require detailed knowledge about the underlying ISO 7816-4 standard, higher-level interfaces are grouped. Examples are *CardFile*, *CardFileInputStream*, *CardFileOutputStream*, *CardRandomByteAccess*, and *CardRandomRecordAccess*. These classes and interfaces approximate the usual Java I/O semantics. Other card services are provided for handling digital signatures and for managing card applets.

## Configuration

The terminals and card services available in the system are statically configured. The configuration data are stored in a so-called "opencard.properties" file, which looks as follows:

```
OpenCard.terminals = com.foo.opencard.fooCardTerminalFactory|fooName|fooType|COM1
OpenCard.services  = com.abc.opencard.abcCardServiceFactory \
                     org.xyz.opencard.xyzCardServiceFactory
```

The entry *OpenCard.terminals* contains the fully qualified class name of the available readers, together with parameters for their instantiation. The entry is used to instantiate the terminals that will then be stored in the card terminal registry. The *OpenCard.services* entry lists the available card service factories. The factories are instantiated and stored in the card service registry.

## Programming Model

The basic programming model of the OCF is illustrated with an example. It shows how to read the contents of the $EF_{ICCID}$ file of a GSM SIM card with OCF. $EF_{ICCID}$ contains a 10 byte identification number, has FID 0x2FE2, is located in the root directory (FID 0x3F00), and is of type "transparent". The code for accessing the card from within an application via OCF is shown below.

```
try {
    // initialize OCF
    SmartCard.start();

    // 1) waiting for a card
    CardRequest cr = new CardRequest(FileAccessCardService.class);
    SmartCard sc = SmartCard.waitForCard(cr);

    // 2) getting a reference to the card service
    FileAccessCardService facs = (FileAccessCardService)
        sc.getCardService(FileAccessCardService.class, true);

    // 3) using the card service
    byte[] data = facs.read(new CardFilePath(":2fe2"), 0, 10);

    // do something with the data...

    sc.close();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        SmartCard.shutdown();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Before OCF can be used it needs to be initialized by executing the *SmartCard.start* method. This triggers reading the configuration file, which contains information about the terminal and service factories. By calling *createCardTerminals*, the card factories are instructed to instantiate the individual *CardTerminal* objects, which are inserted into the card terminal registry. In a second step, the card service factories are inserted into the card service registry. The counterpart to *SmartCard.start* is *SmartCard.shutdown*, which ensures that any system resources allocated by OCF are released again.

After initializing OCF, interaction with the card takes place in three steps: First we wait until an appropriate card is inserted, then we get a reference to the card service, and eventually we use it. In the example, we are interested in a card for which a *FileAccessCardService* is applicable. This is specified with a card request object. The *CardRequest* class allows for a detailed description of the kind of card and service we are interested in. Possible parameters include the desired card service, the terminal in which the card has to be inserted, and the maximum waiting time.

The *SmartCard.waitForCard* method is called with the card request object as a parameter. It waits until the request is satisfied or the timeout is reached. Upon inserting a card, the request conditions are checked one by one. The *CardServiceFactory* objects are queried, whether they know the card type (method *getCardType*) and the requested service (method *getClassFor*). When such a card service factory is found, *SmartCard.waitForCard* returns with a reference to a *SmartCard* object. The desired card service is then instantiated by calling the *getCardService* method on that reference. Internally, the card service registry checks for a card service factory that is able to instantiate the card service (method *getCardServiceInstance*). In the example, the actual usage of the card can now be coded in a single line.

As an alternative to waiting for an appropriate card, OCF provides an event mechanism. Events are triggered upon insertion or removal of cards. Objects which are interested in these events have to implement the *CTListener* interface and have to register with the card terminal registry. If events are detected, the registry calls *cardInserted* and *cardRemoved*, respectively.

## Summary

OCF is organized in a two-layer architecture – abstracting from card reader devices and providing high-level Java interfaces of smart card services. It uses an application-centric model in which the card acts as a passive service provider and applications specify which service is required. OCF is statically configured, with terminal and service factories stored in an configuration file. The approach does not allow for proactive card services that supply their functionality to the outside world. To enable this functionality, the card would have to be queried for its services, which would then have to be made available in some kind of lookup-service.

OCF is designed for usage from within the same Java VM that runs the application. This prohibits the concurrent use of smart cards by different applications. If the reader is connected via the serial interface, for example, the serial port is blocked by the first application using it. Moreover, OCF's mechanisms for mutual exclusion are limited to a single VM.

# *JavaCard RMI*

JavaCard RMI (JCRMI) [16, 17, 18] extends JavaCard with a subset of standard Java RMI. It provides a distributed object-oriented model for communicating with on-card objects. In this model, the card acts as a server that makes remote objects accessible for invocation by off-card client applications. Client applications invoke methods on the remote interfaces of these objects, as if they would be locally available.

JCRMI is built on top of the APDU-based messaging model. It completely hides the APDU layer by automatically encoding (marshalling) and decoding (unmarshalling) method invocations, request parameters, and result values. This relieves the implementer of the card applet as well as the developer of the client application from the tedious tasks of specifying the APDU protocol and coding the parameters manually. As the JavaCard example above shows, if APDUs are used directly, a significant portion of code is devoted to decode incoming and encode outgoing APDUs.

The idea of applying distributed object-oriented and RPC-like principles to smart cards was first implemented by Gemplus. They developed a system called "direct method invocation" (DMI) which was part of their GemXpresso rapid application development environment [19].

## On-Card JCRMI

JCRMI covers on-card programming as well as the client side. The on-card part will be described first.

### Remote Objects

In the same way as in standard RMI, objects that can be invoked remotely are specified in terms of one or more remote interfaces. Remote interfaces extend *java.rmi.Remote*. The methods of a remote interface must declare *java.rmi.RemoteException* in their throws-clause. An example is shown below. It defines the remote interface for the JavaCard example given above. This interface has to be made available to the off-card client application.

```
package prepaidrmi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javacard.framework.UserException;

public interface PrePaid extends Remote {

    public static final short BAD_ARGUMENT = (short)0x6000;
```

```
    public static final short PIN_ERROR = (short)0x6001;

    public void debit(short amount, byte[] pin)
        throws RemoteException, UserException;

    public short getBalance()
        throws RemoteException;

}
```

The limitations of the JCRE and JCVM impose a number of constraints on the type of parameters and return values:

- parameters can only be of primitive type (boolean, byte, short, and optionally int) or single-dimensional arrays of primitive types (boolean[], byte[], short[], and optionally int[]);
- return values can be of any type supported as an argument type, type void, or any remote interface type.

Additionally, functional constraints exist. For example, remote references to on-card objects cannot be passed from the client VM to another VM.

**Remote Object Implementation**

After the remote interface has been designed, it needs to be implemented as a JavaCard class that will later execute on the card. In addition to implementing the remote interface *PrePaid*, class *PrePaidImpl* in the example below extends *javacard.framework.service.CardRemoteObject* and calls its default constructor upon creation. This ensures that the object is "exported" and therefore ready to accept incoming remote method calls. Only exported remote objects may be referenced from outside the card. As an alternative to extending *CardRemoteObject*, *PrePaidImpl* could have called *CardRemoteObject.export(this)* in its constructor. Note that the implementing class does not need to handle APDUs at all. They are completely hidden by the RMI layer, resulting in much cleaner code.

```
package prepaidrmi;

import javacard.framework.service.CardRemoteObject;
import javacard.framework.Util;
import javacard.framework.OwnerPIN;
import javacard.framework.UserException;
import java.rmi.RemoteException;

public class PrePaidImpl extends CardRemoteObject implements PrePaid {

    private short balance;
    private OwnerPIN pin;

    public PrePaidImpl(short initialBalance, byte[] pinData) {
        super(); // export this instance
        balance = initialBalance;
        pin = new OwnerPIN((byte)3, (byte)4);
        pin.update(pinData, (short)0, (byte)4);
    }

    public void debit(short amount, byte[] pinData)
        throws RemoteException, UserException
    {
        if (!pin.check(pinData, (short)0, (byte)pinData.length))
            UserException.throwIt(PIN_ERROR);

        if (amount < 0 || amount > balance) {
            UserException.throwIt(BAD_ARGUMENT);
        }
```

```
        balance -= amount;
    }

    public short getBalance() throws RemoteException {
        return balance;
    }

}
```

**A JavaCard Applet Using JCRMI**

As for non-JCRMI applets, the central class for applets providing remote objects is derived from *javacard.framework.Applet*. The only difference is that APDU objects that are passed in to the *process* method are not handled by the applet itself. Instead, a dispatcher service passes them to an *RMIService* object. The dispatcher is part of a service framework that routes incoming APDUs through a list of services. One after the other, they process the incoming – or later the outgoing – APDU. In the example below, the JCRMI service is the only service registered.

There are two types of APDUs defined in the JCRMI protocol: applet selection APDUs and method invocation APDUs. The JCRMI service handles both of them.

The constructor of the JCRMI service takes a reference to an initial remote object. This will be the first remote object that off-card client applications can talk to. In the example, it is also the only one. Upon applet selection with an appropriate selection command APDU, the JCRMI service returns the object ID and the fully qualified name of the initial reference in the response APDU. Depending on the format requested in the selection APDU, this might either be the fully-qualified class name – *prepaidrmi.PrePaidImpl* in the example – or the fully-qualified interface name, which would be *prepaidrmi.PrePaid* in the example. This information is used as a bootstrap for the off-card client application, allowing it to instantiate the appropriate stub class for talking to the card object.

Incoming method invocation APDUs contain the object ID of the invocation target, the ID of the method to invoke, and the marshalled parameters. The JCRMI service is responsible for unmarshalling the parameters and invoking the requested method of the referenced object. After the remote object implementation has done its work, the RMI service marshals the result value or the exception code, respectively, in the response APDU and sends it back.

```
package prepaidrmi;

import java.rmi.Remote;

import javacard.framework.APDU;
import javacard.framework.ISOException;
import javacard.framework.service.RMIService;
import javacard.framework.service.Dispatcher;

public class PrePaidApplet extends javacard.framework.Applet {

    private Dispatcher disp;
    private RMIService serv;
    private Remote prePaid;
    private final static byte[] pinData =
        { (byte)0x31, (byte)0x32, (byte)0x33, (byte)0x34 };

    public PrePaidApplet() {
        prePaid = new PrePaidImpl((short)1000, pinData);
        serv = new RMIService(prePaid);

        disp = new Dispatcher((short)1);
        disp.addService(serv, Dispatcher.PROCESS_COMMAND);

        register(); // register applet with JCRE
```

```
    }

    public static void install(byte[] aid, short s, byte b) {
        new PrePaidApplet();
    }

    public void process(APDU apdu) throws ISOException {
        disp.process(apdu);
    }

}
```

## Off-Card JCRMI

Having described the card-resident part, we will now show how an off-card client application can use JavaCard objects remotely via standard RMI.

The client application typically runs on the computing device to which the card reader is connected. This could be a stationary PC running J2SE or a mobile phone running J2ME, for example. The client-side API is defined in package *com.sun.javacard.javax.smartcard.rmiclient*. It is designed to be independent from the Java platform and card access technology used. It provides the client application with a mechanism to initiate an RMI session with a JavaCard applet and to obtain an initial reference to the main remote object of an applet. In combination with the on-card service-framework it allows the client to introduce various transport level policies, e.g. transport level security. The main classes *JavaCardRMIConnect* and *CardObjectFactory*, as well as the *CardAccessor* interface will now be briefly introduced.

Interface *CardAccessor* declares method *byte[] exchangeAPDU(byte[] apdu)* to exchange data with the card. Implementations of this interface hide the actual card access mechanism employed. Sun's reference implementation, e.g., runs on J2SE and uses OCF as the card access mechanism. Its implementation class is called *OCFCardAccessor*.

Custom policies, like transport level security, can be realized by extending a CardAccessor class and providing a special implementation of the *exchangeAPDU* method. The custom method first applies the policy to the bytes exchanged, like encryption or adding a MAC (message authentication code), before it calls the *exchangeAPDU* method of the super class. In the same way it decrypts responses or verifies their MACs. On the card this is paralleled by the JavaCard service framework, as discussed above. First, a security service would get access to the incoming data, decrypting them or verifying the MAC, and would then hand them on to the next service in the chain, e.g. the JCRMI service.

*JavaCardRMIConnect* objects are initialized with a *CardAccessor* object. A session with an applet is started with one of the *selectApplet* methods, which have the following signatures:

```
public byte[] selectApplet(byte[] appletAID)
public byte[] selectApplet(byte[] appletAID, CardObjectFactory cOF)
```

The second parameter specifies a *CardObjectFactory*, as explained below. Method *Remote getInitialReference()* is used to get the initial *java.rmi.Remote* reference of the initial object of an applet. The client usually casts this method to the actual object interface, which is interface *PrePaid* in the example.

Extensions of abstract class *CardObjectFactory* are responsible for instantiating client stubs on the client. *JavaCardRMIConnect* uses the card object factory passed to the *selectApplet* method or a default one. The default card object factory simply locates the stub class of the remote object implementation and instantiates it. The name of this class is returned in the remote reference descriptor that is sent in response to the select APDU – *prepaidrmi.PrePaidImpl* in the example. The corresponding stub name is *prepaidrmi.PrePaidImpl_Stub*. Of course, this requires that the stub was previously compiled using the standard RMI compiler "rmic"[2] and that it is reachable via the class path.

[2] "rmic" has to be called with the "-v1.2" option.

An alternative card object factory, called *JCCardProxyFactory*, does not require the availability of a stub class. It generates the stub object dynamically from the list of remote interfaces that the remote class implements. This requires the dynamic proxy generation mechanism of JDK 1.3 or above.

**Reference Implementation of the JavaCard RMI Client-Side**

The reference implementation runs on J2SE and uses OCF for card access. This essentially means that the OCF library files "base-core.jar" and "base-opt.jar" need to be present in the client class path and that an "opencard.properties" file has to be available, which contains the configuration for JCRMI. The main classes of the reference implementation are *OCFCardAccessor* and *OCFCardAccessorFactory*. They reside in package *com.sun.javacard.ocfrmiclientimpl*. *OCFCardAccessor* extends the OCF class *CardService* and implements the JCRMI client-side interface *CardAccessor*. In its *exchangeAPDU* method, it simply passes the unmodified command APDU to the underlying OCF *CardTerminal* and returns the result APDU. Class *OCFCardAccessorFactory* is the factory of the *OCFCardAccessor* card service and has to be listed in the "opencard.properties" file.

**Client RMI Example**

We will now give a complete example of how a client application might access an on-card object using RMI. The example uses the OCF card accessor and is structured like the basic OCF example shown above.

```
package prepaidrmiclient;

import java.rmi.*;
import javacard.framework.*;
import com.sun.javacard.javax.smartcard.rmiclient.*;
import com.sun.javacard.ocfrmiclientimpl.*;
import opencard.core.service.*;
import prepaidrmi.PrePaid;

public class PrePaidClient {

    private static final byte[] PREPAID_AID =
    {(byte)0xa0, 0x00, 0x00, 0x00, 0x62, 0x03, 0x01, 0xc, 0x8, 0x01 };

    public static void main(String[] argv) throws RemoteException{

        try {
            // initialize OCF
            SmartCard.start();

            // wait for a smart card
            CardRequest cr = new CardRequest(OCFCardAccessor.class);
            SmartCard sc = SmartCard.waitForCard(cr);

            // obtain an OCFCardAccessor for Java Card RMI
            CardAccessor ca = (CardAccessor)
              sc.getCardService(OCFCardAccessor.class, true);

            // create a Java Card RMI instance
            JavaCardRMIConnect jcRMI = new JavaCardRMIConnect(ca);

            // select the Java Card applet
            jcRMI.selectApplet(PREPAID_AID);

            // alternative possibility to the previous line:
            // CardObjectFactory factory = new JCCardProxyFactory(ca);
            // jcRMI.selectApplet(PREPAID_AID, factory);

            // obtain the initial reference
            PrePaid pp = (PrePaid) jcRMI.getInitialReference();
```

```
            // get the balance
            short balance = pp.getBalance();
            System.out.println("Balance = " + balance);

            // debiting 15 money units
            byte[] pin = new byte[]{0x31, 0x32, 0x33, 0x34};
            pp.debit((short)15, pin);

            // get the new balance
            balance = pp.getBalance();
            System.out.println("Balance = " + balance);

        } catch(UserException e) {
            System.out.println("Reason code = 0x" +
              Integer.toHexString(0xffff & e.getReason()));
        } catch (Exception e){
            System.out.println(e);
        } finally {
            try {
                SmartCard.shutdown();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

}
```

The program first waits for a card that can be accessed via an OCFCardAccessor. Then, a JavaCardRMIConnect (jcRMI) instance is created and initialized with an OCFCardAccessor. The jcRMI is used to select the applet by its AID and to obtain the reference for the applet's initial remote object. The functionality of the card is subsequently available via local method calls on the *PrePaid* proxy object. All APDU interactions are hidden by the RMI layer. In the example, the jcRMI implicitly uses the default card object factory, called *JCCardObjectFactory*, which requires a stub object. Alternatively, the *JCCardProxyFactory* could have been used, which dynamically creates a proxy object and does not need a stub. The required code is shown in the example, below the line reading *selectApplet(PREPAID_AID)*.

## Summary

Viewing the card as a component in a distributed object-oriented system, such as RMI, has benefits for the JavaCard applet developer as well as for the client application developer. The APDU layer is completely hidden by the RMI middleware and tedious APDU protocol issues no longer need to be dealt with. In this model the client application gets a reference to a local stub or proxy object and calls it like a local object. Although the range of types allowed for parameters and return values is limited, the available types are still sufficient for most JavaCard-based client-server interaction. Extending card accessor implementations opens the possibility to realize custom policies on the transport layer. On the card, JCRMI is accessed using the JavaCard service framework, which allows for flexible handling of APDUs in a chain of services.

# *PKCS #11 Security Tokens*

For the storage of high-quality cryptographic keys and for the execution of cryptographic operations, like digitally signing a document, humans are dependent on computers. Especially for the protection of cryptographic secrets – encryption and signature keys – secure storage is required, since the breaking of such a secret would have potentially hazardous implications.

Smart cards are security tokens that were especially designed for this purpose. However, a smart card is a device resulting from a design trade-off: protection against convenience. Plastic cards were in use before secure chips were available, and smart cards are simply the continuation of this design with additional

features. When one abstracts from the physical shape and restrictions imposed by usability requirements, and concentrates on the functionality as a security device, the notion of a security token emerges, a physical object that serves for security purposes and can be attached to a host system such as a PC. There is a vast variety of ways to connect smart cards and other security tokens to a PC: serial interfaces like RS-232 and USB with varying speed and connectors, Firewire, Infrared, the PCI bus, and so forth. Each of these interfaces requires different device drivers. Additionally, each vendor provides their own programming library.

In order to facilitate application development making use of cryptographic functionality, the "Public-Key Cryptography Standard" [20, 21] was conceived. Part 11 of this standard, called "cryptoki" [22], describes the interface to a generic security token. The description is intentionally very broad, comprising a large variety of data structures, algorithms, and operation modes. Thus, virtually any cryptographic device fits into part of this description and can be made accessible from PKCS #11-compatible software, assuming that the application is sufficiently flexible to adapt to the varying capabilities of different tokens.

PKCS #11 takes on an object-oriented view of a cryptographic token. A token is viewed as capable of storing three kinds of objects: application data, keys, and certificates. Objects are characterized by their attributes and can differ in lifetime and visibility. As an example, consider a symmetric key that is generated during a session of an application with a cryptographic token. During the session, the key is used to encrypt and decrypt data. Before the session is closed, the key can be stored persistently on the token. Alternatively, the key can simply be discarded.

There are a number of products available that come with interfaces that are compatible with the PKCS #11 standard. They include smart cards, USB crypto tokens, PCI extension cards and other varieties. On desktop computers, they can be used to log into the system, sign email, or access restricted web sites. For example, the Netscape/Mozilla browser suite can employ PKCS #11 tokens [23]. Other products, like the IBM 4758 cryptographic coprocessor, are employed in high-security server environments, for example for banking applications.

## *Smart Cards as Distributed Objects*

Employing the CORBA [24] middleware approach to implementing a distributed object system, it is possible to integrate smart cards as well. Smart card services are described just like other objects in the CORBA Interface Description Language (IDL). By using an adapted IDL compiler, such as described in [25], special objects for accessing the smart card are created. Basically, the invocations of operations on such objects have to be mapped to corresponding APDU exchanges. Special care has to be taken, however, regarding the security of the system. For example, it would be a bad idea to transmit a PIN in clear text from a user interface object to the smart card object for user authorization.

## *Smart Card Middleware for Mobile Environments*

In the GSM and the future UMTS world smart cards (as subscriber identity modules, or SIMs) are the fundamental building blocks for user authentication in mobile networks. Besides their primary role of authenticating subscribers of a mobile operator they offer additional functionality which is addressed by smart card middleware. In this section we concentrate on two such technologies: (a) the SIM Application Toolkit (SAT), which offers an application platform in GSM SIMs and which is already used in various operator-based services, forming the basis of the so-called "SAT Browser" technologies, and (b) the current activities in the Java community to specify interfaces between mobile handsets such as mobile (smart) phones and security modules such as SIMs.

### SIM Application Toolkit

The "SIM Application Toolkit (SAT)" [26] is an interface implemented by GSM mobile phones offering among others the following services to the GSM SIM:

- **DisplayText(text}:** Displays the supplied text on the display of the mobile phone.

- **GetInput([title][,type]):** Displays an optional title text and queries the user for input. Several syntactic categories such as digits, hidden input, etc are supported. The text entered by the user is returned to the SIM.
- **SelectItem([title][,item]...):** Displays an optional title and a number of items among which the user can chose. The number of the chosen item is returned to the SIM.
- **ProvideLocalInformation(type):** Return localization and network measurement results depending on the given type selector. In particular it can be used to yield the *network cell identifier* and *location area* information enabling the rough localization of the user's current position.
- **SendShortMessage([title,]dest,payload):** Sends a short message with the given payload to the destination.

Central to the toolkit is the so-called *proactive command manager* which is responsible for managing a *proactive session*. Such a session can be initiated by a smart card-resident applet wishing, for example, to execute the toolkit command *SelectItem*. The applet invokes the appropriate method in the SIM API that in turn activates the proactive command manager that sends a response APDU to the mobile phone in the form of a status word (SW1, SW2) = (91, length). This response code indicates to the mobile station (MS) that the SIM wishes to start a proactive session. The MS then fetches the next command with the given length that contains the proactive command, which in our example contains the items the user has to select from. It then decodes the proactive command contained in the response APDU and in case of a *SelectItem* displays the menu items on the screen of the mobile phone. After the user has selected an item, the MS compiles a so-called *terminal response* APDU that contains among other information the index of the item the user has selected. This response is intercepted by the proactive command manager that in turn resumes the applet and passes the user's selection back to the SIM toolkit application.

Besides the commands listed above, the SIM toolkit further supports a number of mechanisms for registering timers that can be used to wake up an applet at regular intervals, registering for certain types of events such as the arrival of an SMS [27] or a change in the current network cell. The most important triggering mechanisms are the arrival of an SMS and the selection of the applet in the phone's SIM-specific menu.

Summing up, the SIM application toolkit allows to temporarily exchange the role of client (which is now the SIM) and server (which is the MS offering services to the SIM). It can be seen as a platform on top of which card-resident applications can be implemented that have access to an API that allows to perform user interaction and communication.

**A Sample Application**

Several European telecommunication operators run SAT-based services, e.g. in the financial sector. Users can start a session with their preferred bank by invoking their application by selection from their SAT menu in their mobile phone. The card-resident banking applet is invoked and enters a SAT proactive session to query the user for authentication information such as a PIN using the proactive command *GetInput()*. By exchanging encrypted Short Messages (SMS) via some gateway within the operator's infrastructure, the toolkit applet can run appropriate challenge/response protocols with the bank (often 3DES based on [28]) to authenticate users and confidentially transfer financial information such as account balance or trigger financial transactions.

Thus, SIM toolkit is an application platform implemented by the SIM and the MS to offer smart card-based services for applications with high need for security.

**SIM Toolkit Browser Technology**

Although the SIM toolkit technology offers an interesting platform for certain application domains in the mobile world, it suffers from the general problem that for the realization of a particular service an appropriate card-resident applet has to be written and installed on the subscriber's smart card.

SIM toolkit browsers such as the USAT Interpreter [29] try to overcome this limitation by providing a generic platform for the provision of such services by implementing a generic browser application which permanently resides inside the card. Services are implemented with the help of a network-resident gateway component that is in direct contact with the card-resident browser using appropriate communication channels such as SMS or USSD (unstructured supplementary service data).

In the above example of the financial service, the SAT browser would upon invocation dynamically load so-called *pages* that contain in a compact representation the encoding of the SIM toolkit commands it should execute. It then interprets these SAT commands by starting the necessary user interaction and sends the results of the user interaction back to the gateway which takes appropriate action. By exchanging messages between the SIM and the gateway this process continues until the application is finished.

SAT browser technology essentially follows the approach of traditional Web browsers, where the server sends to the client a description of the data to render together with some interaction controls. Some additional primitives are added, e.g. for securing the communication between the gateway, the SIM, and optional third parties such as banks. Commercially available gateways and browsers even go so far that the page descriptions are written in an encoding similar to the Wireless Markup Language (WML) which is then recoded in the gateway into a form suitable for the SAT browser.

Summing up, SAT browser technology directly builds upon the SIM toolkit by providing an even simpler to use middleware for running security-sensitive applications and services in the context of SIMs and mobile networks.

## J2ME Smart Card Middleware

Although SIM toolkit-based solutions potentially offer high security and with the advent of the USAT interpreters also high flexibility in the provision of services, they suffer from the inherent limitation in the communication bandwidth between a smart card and its outer world. It is simply not feasible to digitally sign large documents via a SAT interface that is bound to the length of APDUs in the exchange of information. Therefore, there is considerable interest in bridging the gap between mobile terminals and security tokens such as SIMs by means of appropriate *application-level* interfaces that enable a terminal-hosted application to exploit the security services of smart cards.

Java and in particular the Java 2 Micro Edition(TM) (J2ME) with the Connected (Limited) Device Configuration (CDC/CLDC) emerges as an interesting platform for mobile applications. On the other hand additional card-resident security services such as the WAP Wireless Identity Module (WIM) enter the market. Typically, these are implemented as additional applications on a SIM that are independent from the card's GSM/UMTS core. Since the SIM itself is designed to be one of possibly many applications hosted by the Universal Integrated Circuit Card (UICC) platform, the WIM and possibly other security services could be easily hosted by future UICCs. However, there is currently no standard or interface defined that enables a J2ME application on a mobile terminal to make use of the security services potentially offered by the UICC/USIM.

The Security and Trust Services API specification defines an optional package for the J2ME platform and is currently being drafted by an expert group working on the Java Specification Request 177 [30]. The purpose of this effort is to define a collection of APIs that provide security and trust services to J2ME applications by integrating a "security element". Interaction with such an element provides the following benefits:

- Secure storage to protect sensitive data, such as the user's private keys, public key (root) certificates, service credentials, personal information, and so on.
- Cryptographic operations to support payment protocols, data integrity, and data confidentiality.
- Secure execution environment for custom and enabling security features.

J2ME applications would rely on these features to handle value-added services, such as user identification and authentication, banking, payment, loyalty applications, and so on. A security element can be provided in a variety of forms. For example, smart cards are a common implementation. They are widely deployed in wireless phones, such as SIM cards in GSM phones, UICC cards in 3G phones, RUIM cards in CDMA

phones, and WIM applications in a SIM or UICC card in WAP-enabled phones. For example, in GSM networks, the network operator enters the network authentication data on the smart card, as well as the subscriber's personal information, such as the address book. When the subscriber inserts the smart card into a mobile handset, the handset is enabled to work on the operator's network.

## *JiniCard*

Smart cards are typically used in a variety of places, because they are highly portable and carried by their users wherever they may go. In each environment, a card and its services are present for a short term only and may appear and disappear without prior notice. Yet smart cards depend on their environment to be useful, as they lack an energy supply and user interface capabilities. These usage characteristics call for a middleware layer that provides an effortless integration into different environments without the requirement for any manual setup or configuration. As soon as a card is inserted into a card reader, the discovery of its services should happen spontaneously.

The JiniCard [31, 32] architecture integrates smart cards into Jini federations following the requirements outlined above. Jini's [33] objective is to provide simple mechanisms that enable devices to plug together to form an impromptu community – without any planning, installation, or human intervention. JiniCard makes card services accessible as Jini services. It supports any card that adheres to the ISO 7816-3 communications standard. The main idea of the JiniCard framework is to keep all functionality that is required to interact with a specific card service remotely on the net. It is loaded only when the respective card is actually inserted into the reader. The local environment is thus not forced to know the details of all kinds of cards it might interact with, but just their high-level Jini interfaces.

The JiniCard architecture consists of two layers. The lower layer provides the abstraction of the card reader as a Jini service, in order to make smart cards accessible as network components. The upper layer consists of a smart card exploration mechanism, which identifies the available services. It also provides a well-defined platform for the execution of card-external parts of card services, which are instantiated as the result of the exploration process.

The service exploration process is triggered by reading the ATR of a card. The ATR is mapped to a set of "card explorers" which are dynamically downloaded from the Internet, if they are not yet available locally. The exploration process is then delegated to the appropriate card explorers, which identify the card's services. The result is a set of *ServiceInfo* objects that each contain a code base URL for the service and various Jini-specific attributes. The services are then downloaded and instantiated within the runtime environment provided by the JiniCard framework, which also manages their registration in the Jini lookup service.

From the programmer's perspective, JiniCard provides an extensible framework [34] that helps to easily build the card external parts of smart card applications. New card services can be deployed by uploading new card explorers to a well-known Web server and by providing corresponding card service implementations.

## *Smart Cards on the Internet*

The use of Internet protocols for accessing smart card functionality was proposed for the integration of smart cards into commercial systems on the World Wide Web. The idea is that a Web user points his browser to the items he wants to buy and initiates the payment transaction through a click on a certain URL. This triggers a smart card session to start. The user is prompted to enter his PIN, which authorizes the smart card to carry out the payment. Many payment protocols involving smart cards have been proposed, but here we are interested in the issues that arise when a smart card is interfaced through an Internet protocol. The research projects described were inspired by the availability of JavaCard, which finally turns smart cards into programmable computing platforms.

### A Browser Interface for Smart Cards

The first prototypical system that employed a Web browser to access a smart card was proposed by Barber [35]. A local HTTP server translates requests (in the form of URLs) into APDU exchanges with a smart

card. The type of exchange depends on the path given in the URL. For example, some path triggers the upload of a new applet onto the card, while another path allows the user to directly state an APDU to be sent to the card. Yet another path provides a high-level interface to a specific card service, such as a payment transaction.

Parameters are given in the URL request and are translated into APDUs by the underlying component, called “card server”. The card server can be configured such that it only accepts requests originating from the local machine, thus providing a simple mechanism for restricting access to the smart card to the local user. This approach is quite flexible, since it allows for arbitrary functionality within the card server. For example, it can prompt the user for confirmation, or interact with the keypad attached to the card reader. The card server, as described in the original paper, is itself based on the OpenCard-Framework, which provides the basic smart card functionality.

## Smart Cards as Mobile Web Servers

A different approach is based on the observation that every GSM mobile phone is essentially a portable smart card reader, carrying the SIM for customer authentication. Thus, in many countries the basic infrastructure for carrying out secure business transactions is already in place. However, technical limitations of mobile phones as well as business interests of network operators are obstacles to making this infrastructure openly available. The WebSIM approach [36] tries to overcome these limitations by making a well-defined set of commands residing on the SIM available on the Internet.

The SIM is turned into a Web server through a proxy, which resides on a server on the Internet and accepts HTTP requests and turns them into messages in the SMS (short message service) format. These messages are sent through a GSM gateway to the user’s mobile phone where they are delivered to a JavaCard applet that resides on the SIM. Thus, the applet is able to react to HTTP requests. A result to such a request is returned also via SMS to the gateway and then delivered to the requestor as an HTTP response.

The SIM applet has a number of possibilities of executing requests. First, it can access the SIM’s resources, such as the phone book entries that are stored in a smart card file. Thus, it is possible to update these entries through a Web browser interface without taking the SIM out of the mobile phone. Also, the applet can employ the SIM Application Toolkit to interact with the mobile phone, and thus with the user. For example, it is possible to show a list of items on the phone’s display and ask the user to choose one of them. Another possibility is to use the SIM to create a digital signature on some data that are sent together with the request. Parameters like that, or an item list, are encoded within the request URL.

The address of a WebSIM is composed of a fixed Internet address, which is the name of the WebSIM proxy, and the telephone number of the mobile phone (which is actually the address of the SIM within the GSM network).

## Internet Smart Cards

A similar, but more general approach, is the integration of smart cards at the TCP/IP level [37, 38, 39], which makes it possible to access smart cards directly from the Internet, for example through a Web browser. This is achieved by implementing a simplified TCP/IP stack as a JavaCard applet on the smart card itself. This makes it possible to route TCP/IP packets across the card reader interface to the card.

Webcard, for example, follows a simple protocol. It understands HTTP GET requests, followed by a URL pointing at a smart card file. If the URL path is a valid file name, the content of this file is returned as an HTTP response (the HTTP status line is supposed to be stored in the file together with the actual content). If the named file does not exist, the content of a default file is returned, indicating an error condition.

Impressively, as of the time of writing, a Webcard instance has been running for years and is still accessible on the Web through the URL http://smarty.citi.umich.edu/. A nice feature is the link that points to the source code of Webcard, which is stored on the smart card itself.

Making security sensitive services available on the Internet requires effective access control mechanisms. Data exchange between an application and a smart card often requires confidentiality, for example if the user PIN is presented to the smart card. Webcard does not support such security-sensitive operations.

Instead, this was addressed by another project [38], where a secure link is established between a host application and the smart card before any sensitive data is exchanged. This makes it possible to access smart card services even if the connection between the host and the smart card cannot be trusted to be secure. Thus, applications like SSH and Kerberos can employ the cryptographic functionality of a remote smart card through such a secure link.

## Conclusion

Smart cards are not merely "dumb" (albeit secure) storage devices for cryptographic secrets, but have evolved to "real" computing platforms that are able to host complete and useful applications. This became possible through the progress in microprocessor technology and the availability of Java for smart cards. We hope it has become clear that making use of their advanced features requires sophisticated middleware concepts that are tailored to specific application areas. Many approaches are still evolving, especially in the mobile telecommunication world. Thus, although this chapter on middleware concludes, the story on smart card middleware continues.

## References


[1] International Organization for Standardization (ISO) (1989) *International Standard ISO/IEC 7816: Identification cards – Integrated circuit(s) cards with contacts.*

[2] PC/SC Workgroup (1997) *Interoperability Specification for ICCs and Personal Computer Systems.*

[3] International Organization for Standardization (ISO) (1995) *International Standard ISO/IEC 7816-4: Identification cards – Integrated circuit(s) cards with contacts – Part 4: Interindustry commands for interchange.*

[4] European Telecommunications Standards Institute (ETSI) (1999) *Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface.* GSM 11.11 version 8.1.0 Release 1999.

[5] JavaCard homepage, http://java.sun.com/products/javacard

[6] Sun Microsystems, Inc. (2002) *Java Card 2.2 Runtime Environment (JCRE) Specification.*

[7] Sun Microsystems, Inc. (2002) *Java Card 2.2 Virtual Machine Specification.*

[8] Windows for Smart Cards, http://www.microsoft.com/technet/security/prodtech/smrtcard

[9] MultOS, http://www.multos.com

[10] BasicCard, http://www.basiccard.com

[11] Global Platform, http://www.globalplatform.org

[12] Java 2 Platform, Standard Edition (J2SE), http://java.sun.com/j2se

[13] http://java.sun.com/products/javacard/downloads

[14] *Movement for the use of smart cards in a Linux environment*, PC/SC implementation for Linux, http://www.linuxnet.com

[15] Microsoft Developer Network, http://msdn.microsoft.com

[16] Sun Microsystems, Inc. (2002) *Java Card 2.2 Runtime Environment (JCRE) Specification*, chapter 8: *Remote Method Invocation Service*, pp. 53-68.

[17] Sun Microsystems, Inc. (2002) *Java Card 2.2 RMI Client Application Programming Interface*, white paper.

[18] Sun Microsystems, Inc. (2002) *Java Card 2.2 Application Programming Notes*, chapter 3: *Developing Java Card RMI Applications*, pp. 21-42.

[19] Vandewalle J.-J. and Vétillard, E. (1998) *Smart Card-Based Applications Using Java Card*. Proceedings of the 3rd Smart Card Research and Advanced Application Conference (CARDIS'98), Louvain-la-Neuve, Belgium.

[20] Public-Key Cryptography Standards, RSA Labs, http://www.rsasecurity.com/rsalabs/pkcs

[21] Kaliski, B.S., Jr. (1993) *An Overview of the PKCS Standards*. ftp://ftp.rsasecurity.com/pub/pkcs/doc/overview.doc

[22] RSA Security Inc. (1999) *PKCS #11 – Cryptographic Token Interface Standard, Version 2.10*.

[23] PKCS #11 Conformance Testing, http://www.mozilla.org/projects/security/pki/pkcs11

[24] Object Management Group, http://www.omg.org

[25] Chan, A.T.S., Tse, F., Cao, J., *et.al.* (2002) *Enabling Distributed Corba Access to Smart Card Applications.* IEEE Internet Computing, May/June 2002 (vol. 6, nr. 3).

[26] European Telecommunications Standards Institute (ETSI) (1999) *Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module –Mobile Equipment (SIM–ME) interface.* GSM 11.14 version 8.1.0 release 1999.

[27] European Telecommunications Standards Institute (ETSI) (2000) *Digital cellular telecommunications system (Phase 2+); Technical realization of the Short Message Service (SMS)*. GSM 03.40 version 7.4.0 Release 1998.

[28] European Telecommunications Standards Institute (ETSI) (1999) *Digital cellular telecommunications system (Phase 2+); Security Mechanisms for the SIM application toolkit; Stage 2.* GSM 03.48 version 8.1.0 Release 1999.

[29] 3rd Generation Partnership Project; Technical Specification Group Terminals (2002) *USIM Application Toolkit (USAT) interpreter architecture description*. 3GPP TS 31.112 V5.2.0.

[30] Sun Microsystems, Inc. (2003) *Security and Trust Services API (SATSA) for Java Micro Edition.* Community Review Draft Version 1.0, Draft 0.9, Available at http://jcp.org/

[31] Kehr, R., Rohs, M., Vogt, H. (2000) *Issues in Smartcard Middleware*. Attali, I., Jensen, T., (eds), *Java on Smart Cards: Programming and Security*. Springer-Verlag LNCS 2041, pp. 90-97, 2000.

[32] Kehr, R., Rohs, M., Vogt, H. (2000) *Mobile Code as an Enabling Technology for Service-oriented Smartcard Middleware*. Proc. 2nd International Symposium on Distributed Objects and Applications (DOA'2000), Antwerp, Belgium, IEEE Computer Society, pp. 119-130.

[33] Waldo, J. (2000) The Jini™ Specifications, Second Edition, Addison-Wesley

[34] JiniCard API, http://www.inf.ethz.ch/~rohs/JiniCard

[35] Barber, J. (1999) *The Smart Card URL Programming Interface*. Gemplus Developer Conference, http://www.microexpert.com/smartcardurl.html

[36] Guthery, S., Kehr, R., Posegga, J. (2000) *How to turn a GSM SIM into a Web Server*. IFIP CARDIS 2000.

[37] Rees, J., Honeyman, P. (2000) *Webcard: A Java Card Web Server*. IFIP CARDIS 2000, CITI TechReport 99-3.

[38] Itoi, N., Fukuzawa, T., Honeyman, P. (2000) *Secure Internet Smartcards*. Java Card Workshop, Cannes, CITI Tech Report 00-6.

[39] Urien, P. (2000) *Internet card, a smart card as a true Internet node*. Computer Communications, vol. 23, issue 17, 2000.

Middleware for Smart Cards ..... 1
Introduction ..... 1
ISO 7816 ..... 2
Communication between Card and Card Reader ..... 2
Data Structures on Smart Cards ..... 3
Command Sets ..... 3
JavaCards ..... 3
Hardware Architecture ..... 4
Runtime Environment ..... 4
Developing JavaCard Applets ..... 6
PC/SC: Data Communications ..... 8
OpenCard Framework ..... 10
Architectural Concepts ..... 10
Configuration ..... 11
Programming Model ..... 12
Summary ..... 13
JavaCard RMI ..... 13
On-Card JCRMI ..... 13
Off-Card JCRMI ..... 16
Summary ..... 18
PKCS #11 Security Tokens ..... 18
Smart Cards as Distributed Objects ..... 19
Smart Card Middleware for Mobile Environments ..... 19
SIM Application Toolkit ..... 19
J2ME Smart Card Middleware ..... 21
JiniCard ..... 22
Smart Cards on the Internet ..... 22
A Browser Interface for Smart Cards ..... 22
Smart Cards as Mobile Web Servers ..... 23
Internet Smart Cards ..... 23
Conclusion ..... 24
References ..... 24