Mobile Code as an Enabling Technology for Service-oriented Smartcard Middleware

Roger Kehr
T-Nova Research Labs
Deutsche Telekom AG
roger.kehr@telekom.de

Michael Rohs
Institute of Information Systems
ETH Zurich
rohs@inf.ethz.ch

Harald Vogt
Institute of Information Systems
ETH Zurich
vogt@inf.ethz.ch

Abstract

Smartcards can be seen as service providing entities that implement a secure, tamper-proof storage and offer computational resources which make them ideally suited for a variety of tasks such as authentication, management of personal profiles, and other kinds of secure information processing. Integration of smartcards into networked environments though, has not been achieved yet in a transparent manner.

In this paper we describe the requirements for the design of a middleware for smartcards and propose a platform for the execution of mobile code as the core of such middleware. This is in contrast to traditional architectures based on a request-broker scheme that would need huge standardization efforts to be applicable to smartcards. As an instance of such middleware, we describe our implementation which is centered around the mobile code facilities available in Java and the service trading features of Jini.

1. Introduction

Only recently, large computing systems have begun to show the trend of dissolving into vast crowds of small units, working together in order to supply services to clients. An indication for this development is the extensive proliferation of PDAs and other devices designed for specific, domain-dependent applications, and the increasing need for communication between them. Communication abilities are essential in order to allow for more complex and more useful tasks. Conversely, devices specifically designed for communication, such as mobile phones, are approaching the status of computing platforms (cf. WAP [23]). This development is likely to lead to new computing environments where services are expected to be available at any place, at any time, often referred to as ubiquitous computing environments [25].

In contrast to these increasingly dynamic computing environments, smartcards remain isolated to a great extent, trapped in proprietary application environments. The most attractive smartcard applications, such as authentication, digital signatures, encryption, and cashless payment are far from being universally available. Interworking of devices is one of the main issues in ubiquitous computing, and it is not acceptable that smartcards are not easily integratable into interworking scenarios.

Isolation is even less acceptable in the face of the obvious qualities of smartcards: they are highly mobile, since they are easily carried by their owners, and they are strongly personalized with a focus on security (though not being "perfectly" secure [12], they seem to achieve a proper trade-off between the potential risks and cost). This enables the card to serve as a "user agent" which knows much and reveals little about its owner when acting on his or her behalf. Thus, smartcards could play a vital role in future highly distributed computing systems, under the provision that they are easily integrated into such types of environments. In this paper, we show how such an integration could be approached and what problems are likely to occur.

This paper presents the design of a middleware architecture for smartcards. Jini [19, 21] serves as an exemplary, general service framework upon which we build our architecture. The design of a specialized access point, a smartcard terminal, is presented in Sect. 2. Such a terminal could be implemented as a physical device as well as a virtual device, consisting of a conventional smartcard reader together with supporting infrastructure in the Jini federation itself.

The main functionality of such a terminal is the active exploration of a smartcard's capabilities. This is achieved by dynamically loadable pieces of code, for which Jini supplies a suitable framework. In Sect. 3 we show how smartcard services are integrated in a Jini federation. Sect. 4 describes how application development is facilitated within such an infrastructure.

In the remainder of this section we try to investigate the reasons for the relatively inflexible use of smartcards and

we show how emerging developments could contribute to resolve some of the technical obstacles.

1.1. The Status Quo of Smartcards

We see three main practical reasons for the relative inflexibility of smartcards. First, smartcard interaction is standardized on a per-application basis. Institutional standards exist for (simple) applications in a variety of domains, e.g. data storage, transportation, and mobile telephony [8, 2, 4]. Standards for more sophisticated applications, such as digital signatures, have just recently begun to show up [17]. But standards guarantee interoperability only to a limited degree, as marketing needs require certain variability and proprietary extensions.

The second reason is that smartcards have very limited resources in terms of memory size, computing power, and communication bandwidth. This limits their range of applicability and makes them extremely dependent on their environments. Without an appropriate card reader and a software package, it is impossible for a user to access a smartcard's functionality.

Furthermore, such smartcard environments are often highly proprietary, thus further restricting interoperability. Applications in such environments usually work only with specifically configured smartcards and refuse to interact with third party cards. This has both technical and non-technical, as well as business-related reasons. A prominent example are payment transactions with the German "Geldkarte" (cash card) [3] which are only possible at particular terminals.

This technologically-oriented analysis mirrors the market structure: usually, smartcards are distributed in large quantities and applications are implemented by tight interaction between card manufacturers and card issuers. Hence, these partnerships tend to be very strong and long-lasting, which opposes open architectures.

1.2. A New Age

Smartcards as Software Platforms

Until recently, smartcards and their applications were tightly coupled, resulting in the card being useful for only one application. But the idea of viewing a smartcard as a mere platform has obtained wide dispersion. Increasingly, applications show up that try to use smartcards already in the field for new applications, e.g. ticketing with cash cards [1]. This is possible if the specifications allow for third party access which, however, is usually hampered by strict regulations of the card issuers.

The most consequent step in this direction is the evolving of execution environments for smartcards, such as JavaCard [9] and Windows for Smartcards [26]. This increases extensibility and flexibility and allows to open up the smartcard market to independent application providers. As smartcards are becoming increasingly open, their role as a software platform gains importance and the smartcard paradigm changes, since the separation of cards and applications becomes possible.

The shift towards the platform paradigm is accompanied by a simplification of software development. This is achieved by bringing high-level, standard programming languages to smartcards (Java, VisualBasic), opening up smartcard programming to a new class of developers. Similarly, access to smartcards from applications is unified by architectures like the OpenCard Framework [14] and PC/SC [15], which integrate the card reader infrastructure into operating systems and programming languages.

Nevertheless, in software development for smartcards, manufacturer and application provider independence is still an unmatched goal. It seems reachable, at least.

Smartcards in Networked Environments

In networked systems, devices and applications are working together on different levels. On the network level, protocols are used to exchange data, e.g. TCP/IP. On top of the networking level protocols such as HTTP allow for peer-to-peer communication upon which services can be implemented. On the service level, a server, e.g. a Web server, supports a client in order to provide services to other clients and/or users.

Recent work has shown the feasibility of integrating smartcards on the network level [16, 6, 5, 24]. This makes it possible to regard smartcards as real network nodes.

Integrating smartcards on the service level requires the description of smartcard services, their announcement in a service-trading environment, and establishment of links to the services. These are generic tasks which are usually facilitated by middleware systems. Such a system provides a framework for the description and standardization of services. Lower-level details are hidden in such descriptions, thus standardization can focus on the service descriptions themselves without referring to technical details. Service management is carried out by standard application level services, while communication is performed over standard protocols. This makes access to smartcard services transparent w.r.t. their location and the card's communication features.

Smartcards are devices that are temporarily accessible. Their availability usually corresponds to the physical presence of the user. This requires transparent, quick integration of smartcards into the local environment. Additionally, applications must be designed to handle abrupt disconnections smoothly. Therefore, middleware that offers means to address these requirements is needed.

2. Aspects of Middleware for Smartcards

The goal is to design middleware architectures and systems that facilitate smartcard integration into service federations as much as possible. In the sequel we discuss general design issues of middleware systems leading to our proposed architecure for a smartcard middleware. We start by discussing the basic requirements for a *smartcard terminal*, a component that offers network connectivity for smartcards, and continue by comparing different design paradigms for the middleware implemented in such a terminal.

2.1. Smartcard Terminal

Services implemented in the smartcard must be able to offer their interfaces to the network the smartcard terminal is attached to. Figure 1 illustrates the role of the terminal in a smartcard middleware architecture. A smartcard terminal could be partitioned into the following components:

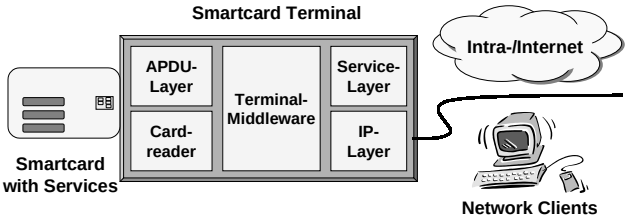

Figure 1. Smartcard terminal

Card reader and APDU layer. The card reader component provides access to the smartcard based on standardized protocols [8]. Essentially, it handles communication between the smartcard and the terminal by exchanging application protocol data units (APDUs, refer to [8] for details).

Network layer. This layer provides basic terminal connectivity to the network. In case of IP this layer would implement an IP stack.

Service layer. This layer presents smartcard services to the network in any suitable form. A number of technologies such as CORBA [13], Java/Jini [19, 21], or DCOM might be used to make the smartcard services accessible from arbitrary network clients. We believe that the actual technology chosen for representing smartcard services should be independent of the scope of the concrete middleware component. Hence, the component should be able to support any of those technologies.

Terminal middleware. The terminal middleware has to perform a number of tasks:

- It must be able to explore the services/applications on a smartcard as it gets inserted into the card reader.
- Based on the service information found it must inform the service layer about the network-interface of the smartcard applications.
- It must act as a gateway for incoming requests from network clients that access the smartcard services via the service layer and forward requests to the APDU layer back and forth.

As such the terminal middleware represents the “glue” between the externally offered network services of the card and the communication layer with the smartcard.

2.2. Design Choices for Smartcard Middleware

Various implementation strategies can be envisioned for the smartcard terminal as outlined above. We describe some possible approaches and compare their strengths and weaknesses w.r.t. the following criteria: *simplicity*, *flexibility*, and *standardization effort* from the perspective of service and application developers on the one hand and middleware implementors on the other hand.

2.2.1 Middleware as an APDU Gateway

This approach can be described as a simple gateway for APDU-requests to the smartcard. Clients send packets to the service-layer of the smartcard terminal containing APDUs which are routed via the APDU-layer to the card reader. Hence, there is no real abstraction above APDUs, and the middleware would be responsible only for multiplexing communication between arbitrary clients and smartcard services.

The interfaces at the service layer would therefore offer methods such as *sendAPDU*, *enterMutex* / *leaveMutex* (needed for locking access to the card for a certain period of time), etc. From the perspective of the middleware implementor this is a rather simple middleware, easy to implement and flexible, since it burdens all the complexity onto the service developer. Services operate at the same level of abstraction as before, but with the intricacies of distributed application programming such as partial failures.

2.2.2 Middleware as Request Broker

With this approach the middleware first explores the services available on the card. This requires an enormous standardization effort since apart from detecting the correct type

of card, there must be a standarized way to perform this exploration. This could be achieved by the definition of new ISO 7816 class and instruction bytes that return descriptions of the services available in the card. Usually, service descriptions consist of interface descriptions, additional infoblocks, and addressing information, e.g. application identifiers, needed to address the service from a smartcard client. This information could come in a variety of formats ranging from binary encoded descriptions to IDL- or XML-based documents.

The middleware could implement a generic server which is capable processing incoming requests from clients and transforms them into appropriate sequences of APDUs. As an example one could imagine a CORBA IDL description that describes a smartcard service which can be used to automatically generate server skeleton code, bind a CORBA object with an ORB running in the smartcard terminal and register the object with a CORBA naming service. In addition to a pure interface description the mapping of method invocations to sequences of APDUs sent to a smartcard need to be defined.

The request broker middleware operates at a much higher level of abstraction. For clients, the smartcard services appear as objects in a distributed object system such as CORBA, Java/RMI, RPC, etc. Service implementors only need to provide an interface definition and appropriate APDU-mappings to integrate legacy applications into the outlined middleware. From the perspective of the middleware implementor numerous standardization steps have to be taken first: exploration of service descriptions, format of descriptions, mapping to distributed object system of choice, service publication, to name a few. We think that this approach though promising in general suffers from the amount of standardization steps necessary for real-world deployment.

2.2.3 Middleware as an Execution Platform for Mobile Code

The middleware architecture presented in this subsection tries to circumvent most of the drawbacks of the previous approaches by completely reconsidering the underlying middleware paradigm. Put shortly, the middleware is not "glue" code between components but a platform for the execution of dynamically downloaded mobile code. We envision the following core scenario:

- The smartcard gets inserted into the terminal and the Answer-To-Reset (ATR) identification string is read.
- The ATR is used to fetch a component that acts as a card manager from a well-known set of Web sites hosting such proxies. These proxies are implemented in a mobile code programming language such as Java. The smartcard terminal provides an execution platform such as a Java virtual machine (JVM). The service proxy is contained in an appropriate Java archive (JAR) file, which is downloaded to the terminal and executed in its JVM. In the basic scenario this card manager could itself now register as a service representing the card with the net.
- In a more advanced scenario the card manager explores the contents of the smartcard in search for smartcard services. This is possible if we assume the implementation of the card manager knows about the particular kind of card which triggered its activation. Hence, it knows how to actually explore the card and find its available services. Each service found may consist of a URL pointing to a service manager which in turn can be fetched and instantiated in the execution platform and offer its particular service to the net.

It should have become clear that this approach essentially defines (a) an execution platform for mobile code, (b) a well-defined process to fetch a card manager from the network, and (c) some API or protocol for the manager to access the smartcard and the network. Compared to the broker-like middleware much less standardization is needed, though the overall flexibility has even increased, since the card and service manager are active components that not only act as services but can also proactively be clients to other services. The most significant drawback with this approach is the fact that the complexity is mostly shifted towards the implementors of card and service managers and the proper definition of an execution platform.

We have found the advantages of the execution platform sufficiently appealing to further experience it by designing and implementing a complete architecture from scratch. In the sequel we describe this architecture along with the most interesting design considerations we were faced with.

3. The Architecture of the JiniCard Framework

As outlined in Sect. 1, smartcards are temporary devices. Consequently, the availability of the services that they offer is short-term and volatile in nature. Smartcards, and hence their services, can appear and disappear without prior notice, that is, *spontaneously*. Smartcards are physically portable and can easily be carried into unknown environments[1]. Yet smartcards are utterly dependent[2] on their envi-

[1] Examples are public and semi-public places like offices, meeting-rooms, banks, post offices, and shops, in which smartcards act as user agents.

[2] For a taxonomy of the design space of small devices along the dimensions of autonomy, computational power, and ability to communicate, see

ronment to be useful, as they generally lack any input or output devices (UIs) for humans. These usage characteristics call for an effortless integration into different environments that do not require any setup or configuration. Service discovery and integration must take place spontaneously.

The requirement for spontaneous integration of smartcards and their services was our motivation to choose Jini as the foundation of the service layer, as described in Sect. 2.1. Jini's objective is to provide simple mechanisms which enable devices to plug together to form an impromptu community, without any planning, installation, or human interaction. Therefore, Jini as a middleware is an ideal choice to support the integration of smartcards, because it meets some essential requirements that are imposed by these ultra-small devices. Jini relies on the Java programming language and the Java virtual machine (VM) as its execution platform. A key point, which we exploit in our architecture, is the ability to move code and objects between physically distributed Java VMs.

3.1. Design Objectives

We call our architecture *JiniCard* to emphasize the fact that it makes card services available as Jini services, independent of the type of smartcard used. It was a key design objective to support a wide variety of smartcards by imposing only a minimal set of requirements on the smartcard's side. Basically, the only requirement on the card's side is that it adheres to the ISO/IEC 7816 standard [8, parts 1–3], i.e. that it communicates by exchanging APDUs, as the vast majority of smartcards does.

One of the main issues that we encountered was how to deal with smartcards that are completely unknown to an environment, given the extremely limited amount of information that can be extracted from an unknown card. A related issue was how to dynamically instantiate card services that are not yet present in the environment at the time of card insertion. In our implementation, mobile code and mobile objects play a major role in this regard. The steps involved in the process of service instantiation will be explained in detail. Finally we will describe what the JiniCard framework looks like for card service developers, i.e. which APIs they can rely on and how they can be used.

Service Integration. An early consideration when developing the JiniCard architecture was that smartcard users are not primarily interested in physical smartcards themselves, but in the services they provide. Therefore, the main goal was to make these services available without much effort on the user's side. Ideally, card services should become part of the infrastructure as soon as the card that carries them is inserted into a card terminal. This should be possible even if there is no a priori knowledge of the services that are contained on a particular smartcard. Another desirable feature, especially if one takes on a more net-centric perspective [7], is to have these services available not only locally, but as part of a local or wide-area network. Therefore the goal can be described as making instances of smartcard services immediately available in a network environment, as a result of inserting a card into a card reader.

The Card Terminal as a Network Component. We felt that the design of current card readers and their device drivers is unsatisfactory to meet these goals. They are usually not self-contained, but attached to a general-purpose PC to function. We propose to view a card terminal as a self-contained entity that provides access to smartcards to a whole network infrastructure. The ultimate vision is to build the JiniCard terminal as a physical device that contains a Java VM, can be plugged into a network, and does not need any additional hardware. This approach requires that such a device is able to describe its capabilities on its own. To make the card terminal available as a network-wide resource, we decided to model it as a Jini service. This has the following benefits:

- the terminal is modeled as a Java interface which means that low level technical details of the implementation of the terminal are abstracted and are no longer important;
- the terminal is seamlessly integrated into an infrastructure and can be used by any client, without any knowledge of the concrete underlying terminal technology; finally,
- the client may be located anywhere in the environment.

3.2. Architecture

The JiniCard framework consists of three categories of components that can conceptually be divided into two layers. The *lower layer* provides the abstraction of a card terminal as a Jini service and serves as a common base for the other components of the framework. The *upper layer* consists of a mechanism to explore smartcards to identify services that are contained on them. The actual *card services* can also be seen as part of this layer. Card services get instantiated as the result of an exploration process. Figure 2 gives a simplified layout of the architecture.

3.2.1 Lower Layer: The JiniCard Terminal

Card services are meant to be downloaded into many different settings. This requires a well-defined environment, con-

[11]. On all dimensions, smartcards rank at the lower end, which means that they are very dependent on proper support from the infrastructure of their environment.

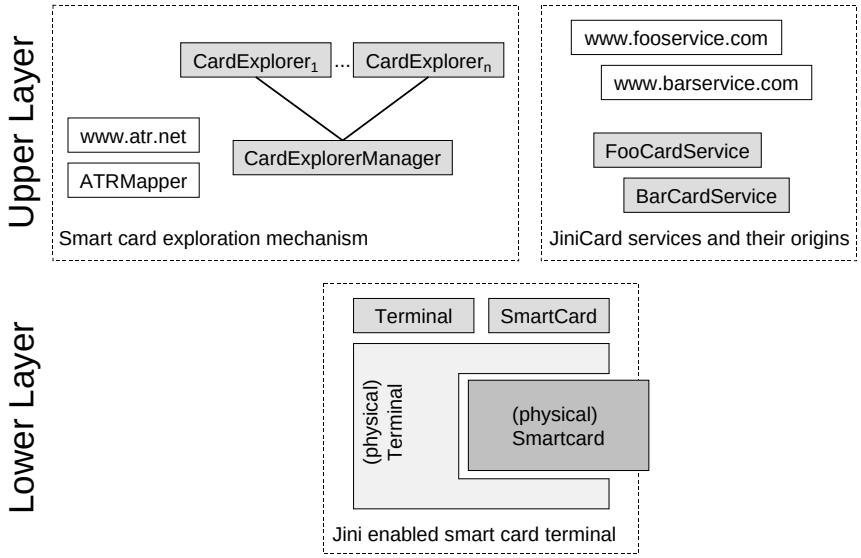

Figure 2. Components of the JiniCard framework

sisting of well-known interfaces, into which these services can be embedded. One way to provide this foundation is by modeling a card terminal as a network component that provides a standard means of remote access to a smartcard.

Accessing Smartcards Remotely. The purpose of the lower layer of the JiniCard framework is to provide a uniform and simple way to access smartcards remotely. With regard to uniform access, motivations similar to those that led to the development of the *Open Card Framework (OCF)* [14] apply here. OCF is a Java-based framework that provides a uniform application interface for building smartcard applications. A major difference to OCF is that the JiniCard terminal is designed to be used remotely and is not restricted to be used by a single Java VM. This means that remote mutual exclusion of access to a smartcard has to be considered.

The card terminal can be assumed to be a more permanent resource than a smartcard, because smartcards are only temporarily inserted into terminals. Therefore it makes sense to consider the card terminal as the foundation of the architecture. Smartcards and the services contained on them are more volatile resources.

We have modeled the JiniCard terminal as an ordinary Jini service. It becomes part of the local Jini federation by finding lookup services and uploading its proxy to them. This process is known as discovery and join [20].

As shown in Fig. 3, the JiniCard terminal has a very thin interface. Using the `notifyStatus` method, clients can register for remote events, which are triggered upon card insertion and card removal. The `getCard` method returns a remote reference to the smartcard that is currently inserted. Calling this method leads to a `CardNotPresentException`, if no card is currently available. Card presence can be tested by using the `isCardPresent` method.

```
package jinicard.core;

public interface Terminal {

    EventRegistration notifyStatus(
            RemoteEventListener listener,
            MarshalledObject handback,
            long leaseDuration)
                throws RemoteException;

    SmartCard getCard()
                throws RemoteException,
                       CardNotPresentException;

    boolean isCardPresent()
                throws RemoteException;
}
```

Figure 3. API of the JiniCard Terminal

In contrast to OCF, the methods for interaction with the actual smartcard are factored out into a separate interface, called `SmartCard`. This interface is shown in Fig. 4. The terminal acts as a resource manager for the smartcard. It is the starting point of access to the card.

It is an inherent feature of smartcards to be available only temporarily and possibly for short periods only. Therefore it is essential to design applications robustly in this respect. The fact that smartcards can be disconnected

without notice is reflected in the design of the `SmartCard` interface. Most methods throw card related exceptions. `CardNotPresentException` is derived from `SmartCardException`, which is the common superclass for all smartcard related exceptions. `CardNotPresentException` indicates that the temporary association between a smartcard and the card terminal has been lost. `RemoteException` indicates that the respective method can be used remotely; it is thrown to indicate (possibly temporary) errors that are related to the underlying communications system. The approach not to try to hide these errors is in line with RMI's general philosophy to make remote exceptions a part of the interface. A discussion of this approach can be found in [22].

```
package jinicard.core;

public interface SmartCard {

    ATR[] getATRs()
            throws RemoteException;

    void setSelectAPDU(byte[] selAPDU)
            throws RemoteException;

    void beginMutex()
            throws RemoteException,
                   SmartCardException;

    void endMutex()
            throws RemoteException,
                   CardNotPresentException,
                   IllegalStateException;

    void reset()
            throws RemoteException,
                   SmartCardException,
                   IllegalStateException;

    byte[] sendAPDU(byte[] apdu)
            throws RemoteException,
                   SmartCardException,
                   IllegalStateException;

    ResponseAPDU sendAPDU(APDU apdu)
            throws RemoteException,
                   SmartCardException,
                   IllegalStateException;
}
```

Figure 4. The SmartCard interface

Maintaining the APDU Interface. The `SmartCard` interface provides a uniform and easy to use abstraction for all kinds of smartcards, but it does not change the basic principles of interaction with a smartcard. The APDU as the low level protocol unit is visible in the interface. A step in the protocol still consists in the exchange of a pair of APDUs – a command APDU followed by a response APDU. This renders the interface very flexible and does not constrain its applicability to certain kinds of smartcards.

Multiple clients of a single JiniCard terminal can hold a reference to the current smartcard simultaneously. Interactions with a smartcard often require the atomic exchange of multiple APDU pairs, e.g. to navigate through a file system hierarchy. During this process state is established in the card. This means that APDUs are not independent of one another. It is not possible to provide transparent scheduling of access to a smartcard, because it is unknown what state was established by one card client, and how to reestablish that state, after another client has been using the card in between. This fact, and the fact that multiple clients can hold references to the same smartcard, requires some kind of mutual exclusion mechanism that is exposed in the interface. This is achieved through the methods `beginMutex` and `endMutex`. They provide mutual exclusion between distributed clients of a smartcard. A problem is that a client can effectively block a smartcard if it does not relinquish control of the smartcard once it has acquired exclusive access to it. Possible reactions to this problem are (1) to ignore it, (2) to use a fixed maximum amount of time that a client is allowed to access a smartcard, (3) to let the client specify in advance (on calling `beginMutex`) how long it needs the card, and (4) to use a fixed maximum inactivity time after which the card is revoked from the client. None of these approaches is without problems, however. For reasons of simplicity, we have chosen the first approach.

A client of the smartcard interface should access a smartcard exclusively only during a single atomic sequence of APDU pairs. Exclusive access should be held as shortly as possible, to give other clients a chance to obtain access to the card.

The method named `setSelectAPDU` is a convenience method for Java cards. Normally, a client cannot assume that a card has not been used by another client between successive exclusive accesses to a smartcard. If another client has used the card in between, it is likely that this other client has changed the state that was established on the card, e.g. by selecting another applet. Therefore a client always has to select its applet again each time it gains exclusive access to the card. With a call to `setSelectAPDU` the client communicated the selection APDU of its card-resident counterpart to the terminal. The terminal records this selection APDU and since it logs all accesses to the card, it is able to decide whether it is required to select the applet again. This is done if meanwhile the card has been touched by another client.

The actual means to talk to the card still is to send command APDUs and to receive response APDUs. JiniCard is fully transparent in this respect. A service implementer can be sure that JiniCard will not change the content of the

exchange of APDU messages. This has the advantage that JiniCard works with all ISO/IEC 7816 compliant cards that rely on exchanging APDUs to communicate.

Immediately after reset, smartcards issue a short sequence of bytes, called the *ATR (answer to reset)*. It contains information about low level communication protocol parameters. It also contains up to fifteen so called *historical characters* that are used in different ways by different vendors. ISO/IEC 7816-3 only states that *"the historical characters designate general information, for example, the card manufacturer, the chip inserted in the card, the masked ROM in the chip, the state of the life of the card. [...]"*. In our approach, we use the ATR simply as a key to obtain further information about a card.

The ATRs of a card are obtained by invoking the `getATRs` method. It returns an array of ATRs to reflect the fact that some smartcards have multiple ATRs. By consecutively resetting a card, it is possible to cycle through the set of ATRs of that card.

A JiniCard terminal service together with the `SmartCard` it manages provides an effective abstraction of the underlying card reader technology. It makes the card terminal and an inserted smartcard a part of the network infrastructure. By modeling the terminal and smartcard as Java interfaces they become easy to use. Clients just need to know the `Terminal` and `SmartCard` interfaces and how to look up a card terminal in a Jini environment. Details related to remote communication are hidden by Jini and RMI. Details concerning the interaction with the physical terminal are hidden by JiniCard. Mutual exclusion allows multiple applications at different locations to act as clients of a single smartcard in an ordered manner. Keeping the exchange of APDUs as the basic means of communication retains the flexibility that is needed to use a wide variety of different smartcards.

As such, the lower layer of JiniCard is an instance of the APDU-gateway middleware described in Sect. 2.2.1 and provides the API for the manager to access the smartcard.

3.2.2 Upper Layer: Smartcard Exploration Mechanism

The components described above provide a uniform way to access smartcards as network components. But they are not enough to reach our goal to effortlessly integrate the services that a smartcard offers into an environment.

To reach this goal, we propose an exploration mechanism to identify the services that are contained on a smartcard and to make them available in the environment. Our approach to reach the goal of card service integration includes the dynamic download of exploration components as well as card-external parts of card services.

As our target environment we chose Jini, which serves as a platform that represents all system entities as services. Therefore we represent all applications contained on a smartcard as Jini services. This places services that are offered by smartcards on an equal footing with other Jini services. In the following sections, we describe the steps that the card exploration mechanism takes.

Smartcard Insertion. The service exploration process is triggered by the insertion of a smartcard into a JiniCard terminal. This causes the terminal to distribute a remote event to all listeners that previously registered with it (1, in Fig. 5). The event contains the ATRs of the card to allow listeners to decide early on, if they are interested in the event and wish to respond to it. The set of ATRs is the only information that can be obtained from a card if there is no a priori knowledge about it.

The Card Explorer Manager drives Card Exploration. The component that controls the card exploration process is known as the *card explorer manager*. This component is registered at the card terminal as an event listener. The card explorer manager manages a set of *card explorers*. Card explorers carry out the actual work of exploring a certain kind of smartcards to identify the services contained on them. Card explorers are dynamically loaded into the Java virtual machine of the card explorer manager, if an unknown kind of smartcard is encountered. The card explorer manager passes a reference to the smartcard on to its card explorers and asks them to explore the card (2). The result of this exploration process is an instance of class `ExplorationResult` (3), which contains a set of `ServiceInfo` objects or an indication that the card explorer could not handle the card. A `ServiceInfo` object describes a single service and provides enough information to engage in the service instantiation process.

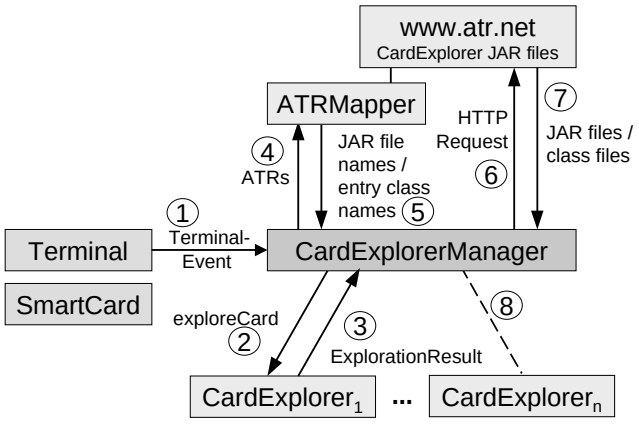

Figure 5. Download and instantiation of card explorers for unknown smartcards

The role of manifest files. What happens if none of the instantiated card explorers was able to handle a card? In this case the card explorer manager contacts a special, well-known Web server. For the following assume that this server is named www.atr.net [3]. This is a Web site that hosts card explorers for many types of smartcards. These card explorers are stored as Java archive (JAR) files. A single JAR file aggregates multiple Java class files and other files. An important part of a JAR file is its manifest file that contains information about the archived files. The contents of an example manifest file are shown in Fig. 6.

```
Manifest-Version: 1.0
Main-Class: jinicard.javacardexplorer.JavaCardExplorer
Created-By: 1.2.2 (Sun Microsystems Inc.)
Smartcard-ATR: O78RAMAQMf5EU01AUlQgQ0FGRSAxLjFDwQ==
```

Figure 6. Example manifest file

Manifest files for card explorers contain two special entries. The first one is the `Main-Class` attribute that was introduced with the Java 2 platform. It allows to designate the class that serves as the entry point into the card explorer. It refers to a class that implements the `CardExplorer` interface as shown in Fig. 7.

```
package jinicard.core.exploration;

public interface CardExplorer {
    ExplorationResult exploreCard(SmartCard sc)
                            throws IOException;
}
```

Figure 7. The CardExplorer interface

The example manifest file refers to a card explorer that is able to explore JavaCards. The second special entry is named `Smartcard-ATR`. Its value is a set of base-64 encoded ATRs. The ATRs have to be base-64 encoded, because the manifest file specification [18] does not allow arbitrary 8-bit entries. This set of ATRs determines the set of cards that the explorer is willing to handle. The example shows the encoded ATR of a JavaCard. This mechanism can be extended by using regular expressions to gain more flexibility. Currently each ATR must be specified separately.

ATR Mapper. A component called *ATR mapper* inspects all card explorer JAR files that are stored on www.atr.net, in order to establish a mapping from a set of ATRs to a set of names of card explorer JAR files.

[3] www.atr.net is just used for illustrative purposes here, so don't worry if it doesn't actually contain card explorers.

If a card explorer manager was not able to find a suitable card explorer for a particular card, it contacts the ATR mapper available on www.atr.net (4, in Fig. 5). The result is (hopefully) the name of a suitable card explorer (5) that the manager can then use for download (6 and 7) and instantiation (8) by using a custom class loader. This newly instantiated card explorer is then in charge of exploring the card in question. Alternatively, the ATR mapper could, instead of a URL, return the actual implementation directly.

Service Information Objects. As already mentioned, the result of a successful exploration process is an `ExplorationResult` instance that contains a set of `ServiceInfo` objects – one for each service. The `ServiceInfo` interface is shown in Fig. 8. An `ExplorationResult` object is what is handed back from a card explorer to the card explorer manager, to enable it to instantiate card services as the final step.

The `ServiceInfo` interface is shown below. It contains Jini related information, such as the *service identifier* (service ID), codebase information and entry point information. The service ID is used to uniquely identify the card service as a Jini service instance. The groups array specifies names of service categories that the service belongs to. *Name* and *comment* are user editable descriptions of a service. The *locators* attribute explicitly specifies lookup services that the service has to connect to once it gets initiated. The Jini specification prescribes that these service attributes (service ID, groups, attributes, and lookup locators) are stored persistently. Once a Jini service gets a service identifier assigned to it, it should remember that identifier and use it in all future interactions with lookup services and other Jini services. To be in line with the Jini specification we decided to store Jini related information on the smartcard whenever possible. For JavaCards for example, we wrote a small applet that stores service information entries by predefined keys. The restrictions in terms of memory space require a clever organization of this information. A further complication is that most entries have varying length and can be changed as a result of user configuration.

JiniCard's Card Services. To enable the card explorer manager to retrieve the actual card service code, the codebase and entry point information are essential. The *service URL* refers to a site that contains the code of the card service described (named www.service.com in Fig. 9). The *service class name* denotes a class that implements interface `jinicard.core.CardService`. With this information the card explorer manager is able to dynamically download and instantiate the card service.

The card service interface is shown in Fig. 10. It serves as an entry point and defines the interaction between a card service and the JiniCard framework.

```
package jinicard.core.exploration;

public interface ServiceInfo {

    // get methods
    ServiceID getServiceID();
    String[] getGroups();
    Name getName();
    Comment getComment();
    LookupLocator[] getLocators();

    URL getServiceURL();
    String getServiceClassName();
    CardService getService();

    // set methods
    void setServiceID(ServiceID sid)
                    throws IOException;
    void setGroups(String[] gs)
                    throws IOException;
    void setName(Name n)
                    throws IOException;
    void setComment(Comment c)
                    throws IOException;
    void setLocators(LookupLocator[] rs)
                    throws IOException;
    void setServiceURL(URL url)
                    throws IOException;
    void setServiceClassName(String scn)
                    throws IOException;
}
```

Figure 8. Interface ServiceInfo

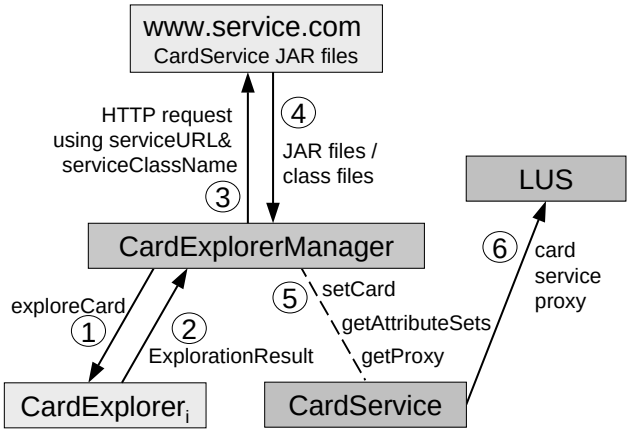

Figure 9. Download and instantiation of card services

```
package jinicard.core.cardservice;

public interface CardService {
    void setCard(SmartCard sc)
             throws SmartCardException,
                    RemoteException;
    Entry[] getAttributeSets();
    Object getProxy();
}
```

Figure 10. Interface CardService

To be useful, a card service must have access to its card-resident counterpart and therefore to the physical smartcard. This is achieved by using the `SmartCard` interface that the JiniCard terminal provides. The JiniCard framework communicates it to the card service by calling the `setCard` method with a remote reference to the smartcard object. It is set to `null` if the card is no longer available.

The `getAttributeSets` method returns Jini attribute sets that are immutable and that do not depend on the specific service instance. The `getProxy` method returns the proxy object that will (in serialized form) be uploaded to the Jini lookup service (abbreviated as LUS in Fig. 9), where it can be downloaded by clients. No restrictions are imposed on the proxy object other than that it is serializable.

This is all there is to know to understand the place that a card service occupies in the JiniCard framework. It uses a smartcard object that abstracts from the need to know anything about the underlying card reader technology or about the location of the smartcard in the network. It interacts with its service manager through the simple card service interface. Everything else is up to the card service developer, who has maximum freedom to design a card service that is appropriate for the application.

4. The JiniCard API from the Service Developer's Perspective

In the following section we will describe how the JiniCard framework looks to the developer, who wants to develop card services using the JiniCard framework.

4.1 Implementing a Card Explorer

If a card service is to be written for a smartcard type for which a card explorer does not yet exist, then the developer has to provide an implementation of the `CardExplorer` interface. This interface has just a single method, named `exploreCard`, that takes a `SmartCard` object as an argument. The card explorer must find a way to explore the set of cards that it is wishes to handle. This can be done by using an on-card directory, which is particularly useful, if multi-application JavaCards are used. Another way to explore a card may be to simply probe the card by using some selection APDUs and by examining if the card generates the expected responses. This is what we have done when implementing a card explorer for GSM cards [2]. The deci-

sion about the way to explore cards is card specific and has to follow pragmatic considerations.

As described above, the result of the exploration process is a set of `ServiceInfo` objects that provide information about a service and also describe how to instantiate it. There are two different possibilities to instantiate card services: One is to provide a URL from which the service implementation can be downloaded (called *serviceURL*), the other is to provide a reference to the card service that the card explorer is able to instantiate by itself. The method `getService` is intended to get a reference to a card service that was instantiated this way. The JiniCard framework first tests if `getService` returns a valid (i.e. non-`null`) reference. If it does not, the `ServiceInfo` object must give a service URL to download the code from. The first approach might be useful if the set of services for a given card is fixed. This allows to store the service implementation together with card explorer implementation. Also, if the card-external code of a smartcard application is stored on the card itself, instead of being stored on a Web server, this might be advantageous.

The reason for making the card-external part of a smartcard application available on a Web server, instead of storing it on the card itself, is the limited amount of memory that is available on current smartcards. The card-external part of a card application may in fact be orders of magnitudes larger than what current smartcards are able to provide. It may, for example, contain a graphical user interface that often needs a large amount of code.

To install a card explorer, all class files that are related to it have to be stored in a JAR file. Its `metainf/manifest.mf` file has to contain the ATRs that are to be handled by the card explorer as well as the name of the implementation's entry class. Finally, the JAR file has to be uploaded to a well-known Web server, like www.atr.net, where it can be inspected by an ATR-mapper.

4.2 Implementing a Card Service

To implement a card service that the JiniCard framework can handle, the following steps must be taken: First, the interface `CardService` (or its subinterface `AdministrableCardService`) has to be implemented. Apart from implementing the interface methods, this means implementing the actual service methods. The service uses the `SmartCard` interface to talk to the card. At runtime, an object implementing this interface will be provided through the `setCard` method. It is important to emphasize that the JiniCard framework does not define the way in which the card-external part of an application talks to its card-resident counterpart. Both parts have to agree upon a proprietary protocol, i.e. a set of APDUs and their meaning. The developer is free to define this private protocol, using APDUs. The developer is also free to design the card-resident part of the application in any way that he or she deems appropriate. This flexibility allows for the integration of cards that provide a fixed APDU protocol. We have, for example, integrated GSM cards into JiniCard that use a standardized APDU protocol that is defined in [2]. In that case the card-resident part, and therefore the APDU protocol, was fixed, and our task was to write a card-external part that integrates a service for GSM cards into the JiniCard framework.

The service related class files have to be packaged as a JAR file and have to be made accessible to an HTTP server. If such a JAR file is small enough, it may also be stored on the card. In any case, the card explorer that explores the card has to be able to examine the service information and to find a way to acquire access to the service code.

If a card service implementation is installed on a multi application card, then its existence has to be announced. This can be done by storing service information in some kind of on-card directory. Card explorers examine this directory to learn about services that are available from the card.

4.3 Implementation and Performance Experiences

The API description in the form of Javadoc pages can be found at [10]. The source code of the JiniCard framework is available from the authors upon request.

Although there is a noticeable delay when the download of a card explorer for an unknown card is required, we found the performance of the JiniCard framework quite acceptable. We expect, that in most cases a card explorer will be available locally and only a card service has to be downloaded and instantiated dynamically. Simple caching strategies could help to improve performance significantly.

5. Conclusion and Future Work

In this contribution we have motivated the need for a new type of middleware that addresses the specific needs of smartcards for integration into a distributed computing environment. Smartcards are one typical instance of small devices with limited computing power and memory resources that pose special requirements to the environment to be useful in a service scenario. These limitations require special attention from the middleware that must be able to integrate devices in a flexible and convenient way.

Our middleware is essentially comprised of an execution platform for mobile code in a card terminal and a well-defined process of how appropriate mobile code is transferred to the terminal as smartcards are inserted into its reader. We argued that our approach outperforms other approaches w.r.t. flexibility and effort of standardization,

which we consider a crucial point in proposing middleware in general.

The Jini network infrastructure has been used both as the trading platform for services offered by smartcards and as a means to implement the JiniCard framework as a set of cooperating network services. We have found Jini to be particularly well-suited for this purpose since it builds upon mobile code, which nicely fits into the paradigm of our proposed middleware.

We think that our approach can be applied to other settings where devices needing assistance from their environments must be integrated into a service federation. Further research into this domain is necessary to support this assumption.

Untouched in our work are security aspects which are especially critical in conjunction with smartcards. Communication between a network client and a smartcard currently traverses several components in the JiniCard framework, i.e. several nodes of different trustworthiness are crossed.

Acknowledgements

We would like to thank F. Mattern, J. Posegga, and U. Wilhelm for many useful comments on earlier versions of this paper.

References

[1] C. Blum. Elektronisches Ticketing bei der Deutschen Bahn AG. In M. Flur, editor, *OMNICARD*, 2000. www.omnicard.de.

[2] European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface (GSM 11.11)*, 1998.

[3] W. Gentz. Elektronische Geldbörsen in Deutschland. *DuD*, 1, 1999.

[4] GSM Association. www.gsmworld.com.

[5] S. Guthery, R. Kehr, and J. Posegga. How to Turn a GSM SIM into a Web Server. In *To appear in Proceedings of CARDIS'2000*, Sept. 2000.

[6] S. Guthery, R. Kehr, J. Posegga, and H. Vogt. GSM SIMs as Web Servers. In *Short-Proceedings of 7th International Conference on Intelligence in Services and Networks IS&N'2000, Athens, Greece*, Feb. 2000.

[7] M. A. Hamilton. Java and the Shift to Net-Centric Computing. *IEEE Computer*, 29(8):31–39, 1996.

[8] International Standards Organization. *International Standard ISO/IEC 7816: Identification Cards - Integrated Circuit Cards with contacts*, 1989.

[9] Java Card Technology. java.sun.com/products/javacard/.

[10] JiniCard API Documentation. Available at www.inf.ethz.ch/~rohs/JiniCard/.

[11] R. Kehr, A. Zeidler, and H. Vogt. Towards a Generic Proxy Execution Service for Small Devices. *FuSeNetD Workshop Position Paper, Heidelberg*, Oct. 1999.

[12] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology*, 1999.

[13] CORBA 2.2 Specification. Available at www.omg.org.

[14] OpenCard Consortium. *OpenCard Framework 1.1.1 Programmer's Guide*, third edition, Apr. 1999. www.opencard.org.

[15] PC/SC Workgroup Specifications. www.pcscworkgroup.com.

[16] J. Rees and P. Honeyman. Webcard: A Java Card Web Server. Technical report, Center for Information Technology Integration, University of Michigan, 1999. www.citi.umich.edu/techreports/reports/citi-tr-99-3.pdf.

[17] RSA. *PKCS #11 - Cryptographic Token Interface Standard*, 1999. www.rsalabs.com/rsalabs/pkcs/pkcs-11/.

[18] Sun Microsystems Inc. *Manifest and Signature Specification*, 1996. java.sun.com/products/jdk/1.2/docs/guide/jar/manifest.html.

[19] Sun Microsystems Inc. *Jini Architecure Specification – Revision 1.0*, Jan. 1999.

[20] Sun Microsystems Inc. *Jini Discovery and Join Specification – Revision 1.0*, Jan. 1999.

[21] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[22] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 1994. www.sunlabs.com/technical-reports/1994/abstract-29.html.

[23] Wireless Application Protocol Forum. www.wapforum.org.

[24] Eurescom P1005 Project. Further information available at www.eurescom.de/~websim/, Apr. 2000.

[25] M. Weiser. The Computer for the 21st Century. *Scientific American*, pages 94–104, Sept. 1991.

[26] Windows for Smartcards. www.microsoft.com/smartcard/.