Issues in 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

1. Introduction

One of the main obstacles to the unification of smartcard usage is the specialization of applications and protocols used between the card and the terminals. Even with Java cards this will not change, since there is no standardized way of exhibiting a smartcard's interface to the outside world. Smartcards communicate by exchanging byte sequences, APDUs, which are only weakly structured. For the ease of parsing, this is often done by using TLV encodings, but in general, each card type defines its own formats.

Traditionally, these formats are standardized by large institutional bodies. However, standardization efforts suffer from their high cost and the rapid change of market needs. Practical interoperability therefore cannot be achieved by standardizing card application interfaces.

Smartcards heavily depend on the environment they are used in. Since these systems are generally equipped with more computing resources than smartcards, the key to interoperability could lie within more sophisticated, generally applicable techniques that help to discover card services and grant access to them. These techniques should support the following issues:

- **Spontaneous integration of cards.** There should be minimal effort required to make an existing system work with new cards. Vice versa, it should be easy to set up a new system using existing cards.
- **Transparent usage of card services.** The system should not have to care about any details regarding the communication to the services that rest on the card.
- **Remote access.** Card services should be available within a distributed environment. For example, the same card, but different services, could be used for terminal login and accounting phone calls.
- **Security.** Only the legal owner of the card must be able to initiate the use of card services. Transactions have to be authorized by the card owner.

This paper shortly describes the JiniCard framework which was developed to enable the seamless integration of smartcards in distributed environments. We give an overview of the system and discuss some security issues in more detail. The system's implementation is available at [4].

2. The JiniCard Framework

In this section, we give a short overview of the system design and show how the requirements are tackled. A more detailed description of the framework can be found in [5]. Security issues are discussed in the next section.

2.1. Design Overview

The JiniCard system is comprised of two major parts. One part consists of a component that directly controls the card reader, while the other part lies within the "net". The core of the card terminal component is the *CardExplorerManager* (CEM). The task of this component is to manage the exploration process of the card. The network part supports the CEM by providing card and service specific components.

Figure 1 shows the components of the system. The lower layer offers basic operations to access card reader functions remotely. Its software parts and the CEM are physically located on the card terminal. All other components reside on remote locations and are requested for code objects only when needed.

2.2. Functional Description

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. There needs to be a mapping between the card and the location where the required code can be found. This mapping is established in two steps.

In the first step, the CEM reads the ATR (or multiple ATRs, if the card issues more than one) of the card and

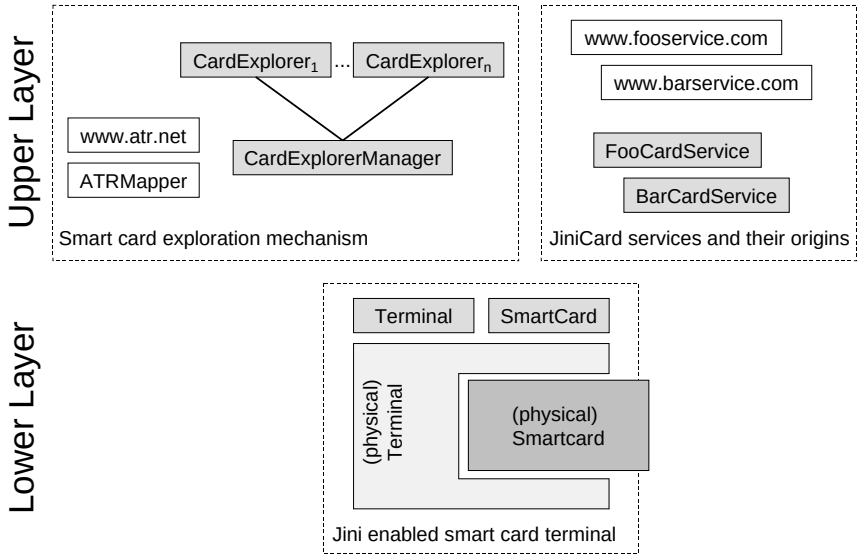

Figure 1. Components of the JiniCard framework

compiles a (HTTP) request which is sent to a server at a well-known address, e.g. *www.atr.net*.

This server returns a set of references to *CardExplorer* (CE) objects which are likely to be able to handle that type of smartcards. Since the mapping is based on the ATR strings only, it might not be possible to reliably determine the proper card explorer. However, this is resolved in the next exploration step

In the second step, all CEs are asked to explore the smartcard. This results in a set of *ServiceInfo* (SI) objects. Each SI object corresponds to a service which resides on the smartcard and for which a CE exists which was able to discover the service. A SI object contains another reference to a piece of code, the respective *CardService* (CS). A CS object is instantiated at the card reader to manage access to the card resident part of the service. Another object, the *ServiceProxy* (SP), can be obtained through the CS. SPs are transferred to the service clients to offer them an interface to the service. They communicate directly to their peer CS objects.

2.3. Embedding in a Jini Environment

The implementation of our framework makes use of the code instantiation facilities offered by Java. The class loading mechanisms offer a convenient method for instantiating objects from (remotely loaded) JAR files. The central server at *www.atr.net* keeps a set of these files, each containing the classes needed by a card explorer.

The CSs are also stored in JAR files and are instantiated through a special class loader which implements further optimizations such as caching of already known card services.

Jini [7] comes into play when card services are to be announced in a distributed environment. Jini offers methods for clients to look up whether needed services are available and also supports many of the administrative tasks linked with the use of remote services (such as timeouts when clients crash or services become unavailable).

The central component of Jini, the Lookup Service, is used to store service proxy objects which implement interfaces to their respective services. Clients send queries to the Lookup Service, asking for proxies which implement a certain service interface. A suitable proxy is downloaded to the client which can use this object to establish a communication to the actual service, which resides on the smartcard.

We assume that the card terminal has sufficient resources to participate in a Jini federation. This requires a relatively large amount of computing and storage capabilities. But since Jini is used only for the announcement of card services, the requirements imposed on terminals could be weakened in more static environments where components are equipped with knowledge about each other.

However, we regard the flexibility of dynamic code loading as crucial in achieving a high degree of interoperability between a wide variety of applications and smartcards. The responsibility of implementing service proxies is shifted to the service implementors who gain freedom in the design of their services. Anyway, the problem of standardized service interfaces remains, but on a much higher level, i.e. on the level of Java interfaces.

2.4. Discussion

The only information that can be reliably used for identification of smartcards lies within the ATR. We use the ATR to roughly determine which card explorers could be suitable for further investigation of the card. Since card explorers are not stored in the card terminal itself but kept remotely on the network, it is easily possible to introduce new cards by providing a new card explorer; no local intervention is necessary.

Transparent usage of card services is provided by service proxies. All details of the protocols used by the smartcard are encapsulated within the proxy. From an application's point of view, the service is represented by a Java object.

Conventional smartcard services are designed to work with a single client. Interleaving card sessions are normally not allowed. As with Java cards, this restriction is eased to some degree.

Within our framework, it is possible for multiple clients to download proxies that talk to a single smartcard. Problems arise when these proxies try to talk to the card concurrently. This happens, when a proxy initiates a conversation with some card applet and sends an APDU to the card, and another proxy does the same right afterwards. The card is then likely to produce no meaningful answer; it is even possible that data will get lost.

To avoid this, we introduced the possibility for proxies to gain exclusive access to a smartcard by acquiring a lock. The card terminal guarantees that no other proxy's APDUs are sent to the smartcard until the lock is released.

A typical interaction with a JavaCard is somewhat different. An applet is selected, then a sequence of APDUs is exchanged. Since there is no (card-) global state information stored within an applet, it poses no problem to deselect an applet at any time and reselect it again to continue the conversation. This enables concurrent usage of different applets on a single card. To handle this transparently to the clients, there exists a convenience method named `setSelectAPDU` in the `SmartCard` interface which is used to define an APDU for re-selecting the applet.

3 Secure Remote Smartcard Access

In the previous section we have outlined an architecture that allows a smartcard to offer services in a network by means of a proactive exploration mechanism initiated by the card terminal. Clients access these card services through Jini service proxies that use the basic interface methods of the card terminal to communicate with the card-resident portion of the service. The most obvious problem with such an approach is the secure access from remote clients to the card, and the problem of the card holder verification procedure (CHV).

3.1 Card Holder Verification

In traditional smartcard scenarios readers are attached to terminals with which the card holder performs CHV to unlock the card for security-sensitive operations. This can be the PIN typed into an ATM, or the PIN entered into a GSM handset to activate the network authentication procedure.

The underlying assumption with this approach is that the communication link between the pinpad or keyboard and the smartcard is secure and cannot be eavesdropped or tampered with. For most of the practical application scenarios smartcards are used in, this assumption sounds quite reasonable. In the scenarios described in our service architecture though, the link between the client and the smartcard must potentially be considered as untrusted and insecure, and special precautions have to be taken.

3.2 End-to-End Security Approaches

The ultimate solution to this problem would be to completely encrypt all information exchanged between the client and the smartcard. This would imply that the traditional ISO 7816 interface based on APDUs cannot be used anymore since it assumes unencrypted APDUs exchanged with the environment. Mechanisms like *secure messaging* [3] are only suitable to encrypt and authenticate the data part of APDUs, but not the class and instruction bytes. Tampering with header information can be detected by adding a digital signature of the header data to the body of an APDU. But not encrypting header information leaves opportunities for eavesdroppers. If the whole APDU is encrypted, then the structure of the APDU becomes meaningless. Such an APDU could only be interpreted after decryption.

In addition to such a completely new card interface we would run into a key distribution problem in case we use symmetric algorithms for encryption. Either the proxy contains the secret key in its state, which could potentially be observed in some runtime environment and exploited to decrypt future sessions with such a smartcard. A better solution would be to use protocols similar to SSL/TLS [1] that use asymmetric encryption to agree on a shared session key between a client and a server. We are in the process of investigating such an approach by implementing the server-side portion of the SSL protocol on a smartcard.

3.3 Secure Session Management via SSL

If we cannot rely on end-to-end secured communication with a smartcard we can require the card terminal itself to be a secure platform. In this case we could, e.g., require any communication from a network client to be transported over SSL. This could be done, for instance, by tunneling Java RMI invocations on top of SSL for which off-the-shelf

solutions are available, e.g. [6]. As a first step, this would offer encrypted communication between the client and the card terminal. Of course we must trust the card terminal not to compromise such an approach. Current practice, e.g. ATM machines, operate with the same trust model. This makes us confident to consider this approach as feasible for many application domains. The main difference between ATM machines and PCs as terminals is that the former are strongly physically secured, whereas the degree of physical protection of PCs is not as high.

Simply encrypting communication to the card terminal is not sufficient, though. It would help to perform a secure remote card holder verification, but leaves the card unlocked for a certain period of time. If an intruder manages to send APDUs to the card via the terminal while the card is unlocked, the system could be compromised.

A solution would be to closely couple operations that require privileged access to the card and SSL sessions. The card terminal that offers access via SSL can uniquely identify peers based on the symmetric session keys used for encrypting and authenticating the communication. When a client performs a card holder verification the card terminal records the peer's identity. Further requests to this card are only allowed from the SSL communication channel that has unlocked the card. This effectively blocks any APDUs sent from other clients until the card is reset again and locked. This is essentially a variant of the semaphore operations `beginMutex`/`endMutex` methods offered by the terminal.

4. Related Work

Closely related to our approach of smartcard integration is the OpenCard Framework [2]. Originally, OCF was designed to run within a single Java VM which would block card readers to other applications. Also, it has no support for remote smartcard access. The proxy concept is used to hide the protocol to the service implementation on the smartcard. Similar to our approach is the use of *CardServiceFactory* objects which produce Java objects through which card resident services can be accessed. Anyway, we found the concept insufficient.

OCF uses an "application driven" paradigm. An application, which runs in the same Java VM as OCF itself, asks for a particular card service and waits until a card implementing that card service arrives. The card remains passive and does not get a chance to announce its capabilities and available services. To achieve this goal, a proactive paradigm is needed, in which the card is asked for its services that are then made available to the environment.

Although OCF defines interfaces and classes for application and card management, they are realized only rudimentary. In particular, the mapping from service descriptions to service instances is not defined.

OCF is a statically configured framework, where all available services must be registered in a configuration file. This is in opposition to the requirement of spontaneous integration that we identified as an important issue in smartcard middleware.

5. Conclusion

We have motivated the need for a standard way of exhibiting a smartcard's interface to the outside world. This issue is still not solved with the advent of Java cards, because the Java card approach does not change the basic means of interaction with smartcards, namely through APDUs.

Smartcards depend on their environments to provide useful services and are also inherently portable by their human owners. Given these characteristics we have identified four key areas, which need to be taken into account by middleware that aims for the interoperability of smartcards. These are spontaneous integration into environments, transparent usage of card services, remote access to card services, and security that is effectively controllable and observable by the card's owner.

We have given a brief description of the JiniCard framework, which aims at the seamless integration of smartcard services into distributed environments. JiniCard relies on the dynamic download of code to identify and instantiate the services that are available on a smartcard. It provides a well-defined platform for the execution of card-external parts of card services.

To achieve spontaneous integration into environments we chose Jini as the underlying middleware technology, because Jini's objective expressly is to provide simple mechanisms which enable devices to plug together to form an impromptu community.

Our approach is easily extensible by uploading new card explorers to a well known web server and by providing card service implementations. It also handles mutual exclusion of multiple clients that try to use a card concurrently. The independence of applets on Java cards seems to make a relatively transparent scheduling approach possible.

Accessing smartcards remotely poses new security issues. In particular, the assumption that the communication between clients and card services is secure no longer holds. Communication between clients in the net and smartcard services can potentially be observed (eavesdropping) and even altered (tampering). Therefore it is considered problematic to completely unlock a card via card holder verification. New approaches have to be investigated to ensure the security of card-to-service communication. One solution that we are currently investigating is to implement the server-side portion of the SSL protocol on a smartcard or on

the terminal attached to the smartcard reader. This makes it possible to geographically separate a card from its client.

Acknowledgements

We would like to thank Joachim Posegga for inspiring discussions at the beginning of the JiniCard project.

References

[1] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Internet RFC 2246, Jan. 1999.

[2] U. Hansmann, T. Schäck, F. Seliger, and M. S. Nicklous. *Smart Card Application Development Using Java*. Springer-Verlag, 1999. See also www.opencard.org.

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

[4] JiniCard. http://www.inf.ethz.ch/~rohs/JiniCard/.

[5] R. Kehr, M. Rohs, and H. Vogt. Mobile Code as an Enabling Technology for Service-oriented Smartcard Middleware. In *The 2nd International Symposium on Distributed Objects and Applications (DOA)*. IEEE Computer Society Press, 2000. http://www.inf.ethz.ch/department/IS/vs/publ/papers/jinicard.pdf.

[6] A. Popovici. ITISSL - A Java 2 Implementation of the SSL API based on SSLeay/OpenSSL. http://www-sp.iti.informatik.tu-darmstadt.de/itissl/, 1999.

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