

# Secure Collaborative and Distributed Services in the Java Card Grid Platform

Serge Chaumette

*LaBRI, UMR CNRS 5800  
Université Bordeaux 1  
351 cours de la Libération  
33405 Talence, FRANCE  
chaumett@labri.fr*

Achraf Karray

*LaBRI, UMR CNRS 5800  
Université Bordeaux 1  
351 cours de la Libération  
33405 Talence, FRANCE  
karray@labri.fr*

Damien Sauveron

*XLIM, UMR CNRS 6172  
Université de Limoges  
123, avenue Albert Thomas  
87000 Limoges, FRANCE  
sauveron@labri.fr*

## ABSTRACT

*Ensuring the security of services in a distributed system requires the collaboration of all the elements involved in providing this service. In this paper we present how the security of collaborative distributed services is ensured in the Java Card<sup>TM1</sup> Grid project carried out at LaBRI, Laboratoire Bordelais de Recherche en Informatique. The aim of this project is to build a hardware platform and the associated software components to experiment on the security features of distributed applications. To achieve this goal, we use the hardware components that offer the highest security level, i.e. smart cards. We do not pretend that the resulting platform can be efficient, but we believe that it is a good testbed to experiment on the security features that one would require for real distributed applications. The kind of applications (and the services they use) that we run on our platform are those that require a high level of confidentiality regarding their own binary code, the input data that they handle, and the results that they produce. This paper focuses on the collaboration aspect of the secure services in our platform.*

**KEYWORDS:** Smart Cards, Java Cards, Distributed Services, Collaboration.

## 1. INTRODUCTION

Because of the development of the technology, the users are asking more and more in terms of computing resources and networks capabilities (bandwidth, mobility, etc.). Moreover, these services should be able to collaborate together to achieve the best results for the end users. To satisfy all these requirements, the manufacturers have developed new technologies to connect the resources (WiFi, Bluetooth, etc.)

<sup>1</sup>Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors are independent of Sun Microsystems, Inc. The other marks are the property of their respective owner.

and to support the development of applications (Java, .NET, etc.). Thus, by federating such shared resources, a user can have access to a large platform (*e.g.* a grid [2, 5]) to execute his services.

However, potentially unknown persons could be authorized to execute their services on such a platform, and the users of such systems must accept to have their services executed on resources that are under the control of someone else who they potentially do not even know. Therefore security is a big concern. First, the owner of the code or more precisely the code itself must be protected from the platform that executes it and from other services executed on the same platform. Second, the computing resource that runs the code must be protected from this code. Even though there are software and hardware level protections, it is clearly not sufficient. If someone uploads a code to my workstation so that it is executed, nothing can prevent me from dumping the memory to work out what it is doing, or even from tracking the instructions executed by my processor. If I upload a code to the machine of someone else, nothing will prevent my code from doing malicious operations, even though sand box approaches can solve some of the problems.

Smart cards [16] provide solutions, at both hardware and software levels. At hardware level, the cards are built so as to resist any physical attack. Of course, attacks remain possible but they will not be feasible in a reasonable amount of time. The processors that can be found in standard workstations do not offer the same protections. When a code is loaded inside a card, it can neither damage the card or access the assets that it contains, nor can it be reverse engineered by the owner of the card. At the software level, the cards and the applications that they embed are evaluated and certified by well defined procedures (*e.g.* ITSEC - Information Technology Security Evaluation Criteria - or CC - Common Criteria) in government approved agencies or companies (*e.g.* ITSEF - Information Technology Security Evaluation Facility).

Furthermore, even though the cards are not very efficient in terms of computation power right now, the resources

that they provide in terms of memory and computing capabilities [17] have increased a lot. Cards that will provide 1 Gigabyte of memory and more efficient processors are expected as soon as 2007.

Therefore we have begun the Java Card Grid project. Within the framework of this project we have designed and implemented a platform that can be viewed as a grid of smart cards. As of today, we have 32 card readers that are connected together. The goal of this platform is to experiment security features that will help in supporting or even designing secure real size grids and the services running on top of them. This platform and more precisely the collaborative services that it supports are the topic of this paper.

Before describing in section 2 the services available in our platform, the most relevant related projects about on-card services are presented and compared with our own approach. Then, in section 3, we describe the overall Java Card Grid platform at both hardware and software levels. Section 4 focuses on how, in our platform, the services are implemented, published to the rest of the world, and then used by a client application or by another card. Then, in section 5, we present the collaborative approach between cards that make then possible to provide altogether a given service and construct higher level applications by composing a number of such services. We eventually conclude in section 6 on the future evolutions of our platform.



Figure 1: The Java Card Grid platform.

## 2. RELATED WORK

We have identified a number of projects, the aim of which is to integrate Java Cards and the services they provide in a distributed environment. The goal of all the frameworks that we have studied is to integrate the on card services as

standard services: Corba[13] for ORBCard [3]; Jini[18] for JiniCard [11]; RMI[7] for JCRMI [12]. Once integrated, the functionalities offered on card become transparently usable from the outside.

It is clearly not the goal of these approaches to use the cards to run CPU demanding applications, or even really distributed applications. They are more designed in terms of local services. On the contrary, we intend to use the cards as cooperating computational resources to support distributed services that we call *distributed inter card services*. To achieve this goal our framework makes the cards proactive, and this is one of the major originalities of our work. Moreover, in our platform, both computation (*i.e.* service execution) and inter card communication are secured. In the rest of this section, we describe the most relevant environments that compare to our work.

### 2.1. JiniCard

The JiniCard [11] approach consists in integrating the services of a smart card into a Jini environment. The card is supplemented with off card services included in a mobile code, following the Jini approach. One of the main advantages is to overcome the static characteristics of the card. Any Jini application will be able to discover and use the services of the card in a spontaneous way. The JiniCard architecture is based on a software component called the CardExplorerManager (CEM), the aim of which is to explore the services included in the card. Once these services have been discovered, objects representing them will be registered in the lookup server. Any object will then be able to discover these services (thanks to the lookup) and to call them. This architecture makes it possible for a card to expose its services to the other components of the network.

Contrary to our approach, a service cannot be implemented and distributed on several cards with a card calling a sort of microservice on another card (*i.e.* to delegate some task to the other cards).

### 2.2. ORBCard

The ORBCard [3] platform is designed to integrate smart cards into Corba based distributed systems. The central element of this architecture is the ORBCard adaptor, which is a bridge between the card and its external environment. The ORBCard adaptor makes it possible for Corba objects to communicate with the services present on a smart card in a transparent way, as if they were communicating with an ordinary Corba object. The ORBCard adaptor transforms the request to an appropriate APDU representation. To communicate with an application of the card, the Corba object in the client application contacts the ORBCard adaptor through the Corba bus. The ORBCard adaptor transforms the request to its APDU representation and sends it

to the application loaded on the card. After the execution of the command, the adaptor receives a response from the card, it converts it into a Corba answer (simple type: void, int, byte, etc.) and transmits it back to the Corba object of the client application.

As it is the case with JiniCard, ORBCard does not allow to implement distributed inter card services.

### 2.3. JCRMI

In Java Card 2.2 [12], Sun includes a new way of communicating with applets: Java Card Remote Method Invocation (JCRMI). From a practical point of view, the object in the card must be described by an interface that extend the `java.rmi.Remote` interface. A client gets access to this object when selecting the applet, getting back a JCRMI reference to the object. To invoke a method on such an object, the client sends an APDU command that contains the identifier of the object, the identifier of the method, and its parameters if any.

As in the previous solutions, the card is not proactive and it cannot propose distributed inter card services.

## 3. THE JAVA CARD GRID PLATFORM

The goal of the Java Card Grid project is to provide a hardware platform, a software framework and the associated administration tools, to deal with a (large) number of interconnected smart cards.

### 3.1. Hardware Platform

As illustrated figure 2, the hardware platform that we have set up contains two grids that are connected together by the network.

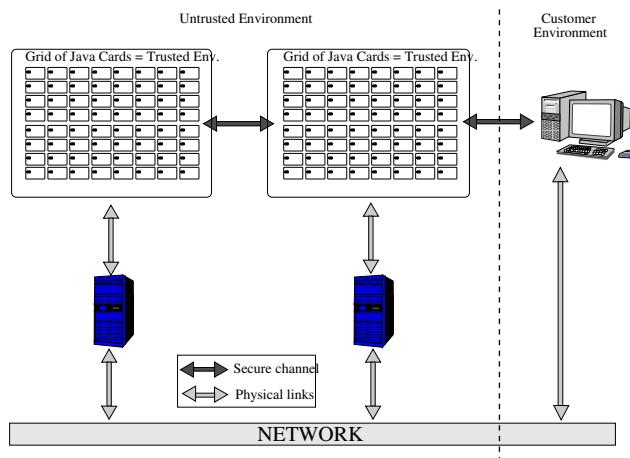


Figure 2: A hardware platform based on Java Card grids.

The hardware fits in a wall mount cabinet of 19U. Each grid is composed of:

- a PC which needs 2U;
- two 2U racks from SmartMount, each one having 8 CCID readers from SCM Microsystems, *i.e.* we have a total of 16 CCID readers;
- three USB 7-port hubs (placed in a empty 2U rack) to connect the readers to the PC and to power the readers;
- Java Cards of different manufacturers plugged in the readers which then power them.

We have also equipped one of the PCs with a LCD monitor and a special rackable keyboard (with an integrated touch-pad) that we use to control the servers. A picture of this platform is shown figure 1.

### 3.2. Software Framework

The software framework that we have designed and implemented comprises two layers: a low level layer that uses PC/SC<sup>2</sup> and that handles the PCs, the readers and thus the smart cards, and a high level layer that manages the distributed computing framework that we offer.

### 3.3. Administration Tools

We have begun to design and implement tools to support remote administration of the grid of Java Cards, *i.e.* to monitor the topology of the grid, to detect the defective cards, to deploy new applications, etc. As of writing, a remote topology control tool (*i.e.* to see which readers are free and which ones contain a card, to track the evolution of the grid, etc.) is available.

The tools and APIs that we are currently working on are dedicated to the automatic and dynamic deployment of applets, and thus of services, on a set of Java Cards. Since most of the Java Cards are GlobalPlatform<sup>3</sup> [6] compliant, we only need to develop an implementation that uses this standard to be able to load and delete embedded applications. We plan to use the open source GlobalPlatform library [14], recently developed by Karsten Ohme, to add this functionality to our platform. With this tool, we will be able to install, delete, and manage services on the cards.

## 4. SERVICES IN THE JAVA CARD GRID

By using the features provided by our communication stack [4], it is now possible to easily and quickly develop

<sup>2</sup>PC/SC [15] is a standard that provides a high level API to communicate with smart card readers.

<sup>3</sup>GlobalPlatform is a standard that specifies APIs to manage multiapplication cards.

high level services (or applications) that will be embedded in the cards of the grid. In our framework, a Java Card is seen as a container for services. The provider develops the applet implementing the service and describes its interface using XML. This is then uploaded to the card. Potential clients are provided with a list of the available services. For this purpose, the software framework initially scans the grid, and extracts the services available on each card. The client can then select and remotely use one of them. In the rest of this section, we get back in more details to these different phases, which are shown figure 3. We eventually illustrate the whole process by means of an example.

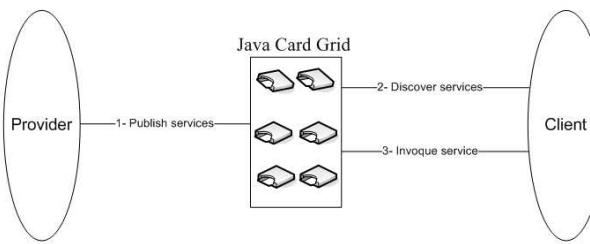


Figure 3: Main steps to communicate with a service.

#### 4.1. Publication of Services

The provider uploads the implementation of its service to the card and then publishes its descriptor in the *ServiceCatalog* application, a local registry, which is present on each card. This operation is done by sending the service descriptor to the *ServiceCatalog* application. The information contained in the descriptor is mainly composed of the identifier of the applet, a description of the operations that it provides and their parameters if any. This information is formatted using XML. The *ServiceCatalog* application maintains and publishes the list of descriptors of the available services.

#### Service Descriptor

We have chosen XML to describe services. The descriptor comprises the identifier of the service, the operations that it offers and their parameters if any. The descriptor of a service is generally smaller than the implementation of the service itself. It is then cheaper to distribute and store. A service descriptor precisely contains:

- The identifier of the service. Each service in the card must have a unique identifier. It corresponds to the identifier of the applet that implements this service (*i.e.* the AID).
- The name of the service.

- The methods of the service. The *method* elements describe the different operations offered by the service. Each service must provide at least one operation. For each operation, the descriptor gives its name and the value of the *INS* byte of the corresponding APDU command. The *arg* tag describes the possible parameters needed by this operation. The *type* tag is the return type of the method. This can be any supported primitive data type (*i.e.* byte, short, boolean and void) or String. The String type is not supported by the cards, but it is translated to a byte array on the client side. Each method description can have a *documentation* tag that can be used to describe the operation.

```

<applet name="HelloWorld" id="11223344556611">
  <method>
    <name>sayHello</name>
    <ins>02</ins>
    <arg>byte</arg>
    <type>String</type>
    <documentation>My first Card Service</documentation>
  </method>
</applet>
  
```

Listing 1: Service Descriptor.

The descriptor shown Listing 1.1 describes a service called *HelloWorld*, identified as service number 11223344556611. It contains only one operation called *sayHello*, which is internally known by the service as operation 02. It requires one parameter of type *byte* and it returns a type *String*. A short description of this operation is given in the *documentation* tag.

#### 4.2. Service Discovery

The service discovery phase can be broken into two stages. The first stage consists in scanning the grid to discover the cards, and the second stage in scanning the cards to discover the services that it contains.

#### Scanning the Grid for Cards

During this stage, we establish the state of the grid, *i.e.* we detect the cards that are present and the empty readers. For each reader which is found, an instance of the *CardProxy* class is created (*cf.* figure 4). The *CardProxy* maintains all the information about the reader it is in charge of, and about the card that it possibly contains. It knows the name of the reader (provided by PC/SC), its state (card present or not) and the name of the card (contained in a specific applet present on the card) if present. The *CardProxy* is a sort of bridge between the client applications and the Java Cards. Thus, all communication with a card will go through the corresponding *CardProxy*. Once the initial state of the grid has been established, any change is detected. For this

purpose, the scanning layer contains a listener thread that detects significant events, such as the insertion or tearing of a card.

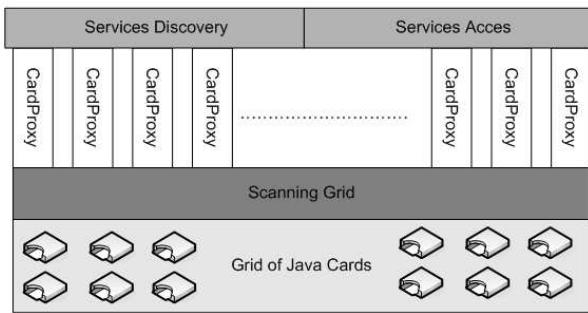


Figure 4: The services management layers of the Java Card Grid.

The insertion and tearing of a card are handled as follows:

- When a card is removed, the grid server that hosts the *CardProxy* notifies the client applications that were connected to this card. The client applications can then take the appropriate measures.
- When a card is inserted, the scanning layer notifies the service discovery layer which then explores the card to extract the list of available services as explained below.

## Scanning the Cards for Services

Each card that is found in the previous stage can now be examined. A query is sent to the *ServiceCatalog* applet to request the list of available services. The response contains all the service descriptors stored in the card. A list of all the services available in the grid, their localization (*i.e.* the name of the card and the reader where it is inserted), and their descriptors can then be built. It will be kept up to date when cards are inserted or removed. This list will be passed to clients when they connect to the grid server.

## 4.3. Accessing Services

As soon as a client connects to the grid server, he is provided with the description of all the available services, and with their localization. He can then choose the service that he wants to invoke by using a dedicated graphical interface. At that point, he only needs to provide the parameters required for the execution of this service.

The services access layer, located on the client machine, will achieve the following operations to invoke the selected service:

1. Select the applet (*i.e.* the service). As the names of the reader and the applet are known, the client application sends a *select* APDU command to the concerned *CardProxy*. The *CardProxy* will simply forward the command to the card.

2. Establish a secure channel between the client application and the service. To achieve this goal, we have designed our own cryptographic protocol [10]. It relies on the exchange of challenges that are used to generate session keys, based on static keys known *a priori* by the two entities.
3. Convert the provided parameters to bytes in order to be able to build the APDU command which will be sent to the card.
4. Effectively invoke the operation chosen by the client.
5. Extract the result from the response APDU that the client application receives from the card when the execution of the command is finished.

## 4.4. A Service Example

The example that we present in this section is a toy application that we have kept simple for the sake of illustration. It shows the implementation and usage of services but does not really take advantage of the grid hardware architecture. It is an internet bookmarks management application for Java Cards. It provides an easy way to store, retrieve, query and carry URLs on the move. The service makes it possible to store a new URL and an associated name, to delete a URL, to search for a URL based on a part of it or on a part of its name, and to retrieve the URLs contained in the card. We first wrote the applet that implements the service, we then wrote the XML descriptor for this service (a part of which is shown Listing 2), and we installed both the applet and the descriptor on the cards of the grid.

```
<applet name ="MyFavorites" id="112233445501">
[...]
<method>
  <name>searchURL</name>
  <ins>06</ins>
  <arg>String</arg>
  <type>String</type>
  <documentation>Search an URL</documentation>
</method>
[...]
</applet>
```

Listing 2: MyFavorites Service Descriptor.

The service can then be used for instance with our client side generic tool (*cf.* figure 5). This tool uses the XML description of a service to automatically build a graphical interface that makes it possible to use it (*i.e.* input parameters, display result, etc.).

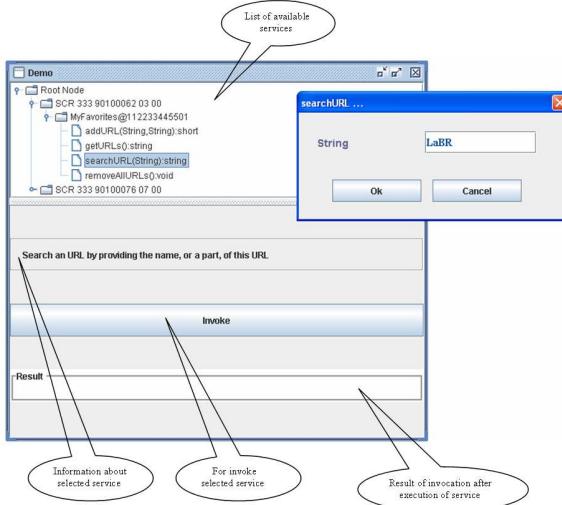


Figure 5: Using the service example.

Note that the URLs and their names are represented by *Strings* on the client side, but they are converted into an array of bytes to be stored in the card, since the *String* type is not supported by Java Cards.

## 5. COLLABORATION OF SERVICES

Most of the time, the components of a distributed system need to communicate with each other to achieve some sort of cooperation. Therefore offering a framework that makes the cards active, *i.e.* able to take the initiative of a communication (possibly with an other card), is mandatory. When such a mechanism is available, the nature of the collaboration for the services between the cards can be double. First, the overall execution of *a single* service can be distributed over several cards. In this case the different parts can be viewed as microservices for the global service itself. We call such a service, a *distributed inter card service*, because its behaviour depends on the cooperation of all the cards. Second, the collaboration of services can also exist between cards themselves when a card calls one or more services on one or more other cards. In this case, the card can do what we call a *composition of services*. The difference between the two cases is that the first is at low level to achieve one service, and the second is at high level to achieve a complex task composed by several basic services (see below).

Of course, an *external* client of our platform can compose services itself.

### 5.1. Proactivity mechanism: the heart for the collaboration

Smart cards are passive. They work according to a master/slave model. The host application (*i.e.* the application which is out of the card) is the master, and the card application is the slave that provides the service. The card is always waiting for a command, and never takes the initiative of a communication with the outside. Because of this passive mode, the card can neither explore its environment nor initiate any interaction with external components or services. To make the card active, *i.e.* able to send a request, for example to invoke a service on an other card, we have set up a simple mechanism which consists in asking the card if it wishes to send a request. To achieve this goal (as illustrated figure 6) an APDU command is sent to the card and if the card has a request to send, it puts it in the APDU response. The mechanism is similar to what is used for SIM cards

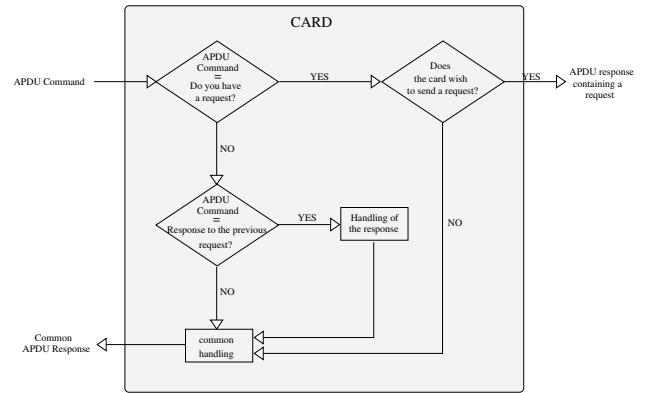


Figure 6: Execution model of a proactive card.

in cellular phones [8, 9]. When an APDU response is received, it is handled by the PC (and more precisely by the CardProxy) which acts as a router. If the response matches a specified format (not detailed here, but which contains all the information to locate a precise service on a precise card of the grid), it means that it is a method invocation of a service located on another card. With this information, the PC forwards the request to the target card. The APDU command is also contained in the APDU response of the client card.

To simplify the calls, we have set up a *Stub/Skeleton* mechanism generated from: an interface which represents the service; its AID; the name of the card where it is installed. Figure 7 illustrates the remote invocation process between cards.

Nevertheless, due to their passive nature, the cards are only able to execute code between an APDU command and the associated APDU response (excepted the waiting loop for the next APDU command). According to our model, if a

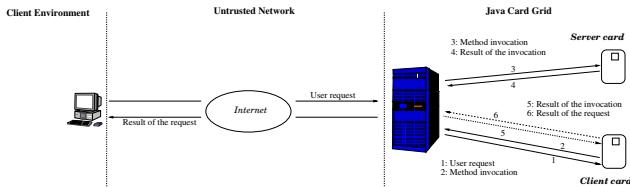


Figure 7: Remote method invocation between cards.

card calls a service (*i.e.* if it sends an APDU response containing a request), its execution (from the application level point of view) is stopped. Thus we have a model with synchronous method calls that can be unsuitable for some kinds of applications.

Moreover a related problem is to continue the execution of the code at the correct point. Indeed, Java Card respects a model where all the APDU commands are handled by the environment of execution (so-called JCRE) before they are forwarded to the `process` method of the *applet* (the unique entry point of a Java Card application).

Finally, it is not possible to receive an APDU command during the execution of an applet, *i.e.* while an APDU command is being processed and before its associated APDU response is sent back by the card. This is due to the fact that the card is a mono-threaded server that can serve only one request at time, without the capability to queue any other call.

Due to all these constraints, an *applet* cannot call an outside service and wait for the response (that would be a second APDU command). Therefore, to make the cards proactive, we have also set up a mechanism that makes it possible to continue the execution at the correct point when the result of the service invocation comes back. Thanks to our mechanism, when a *client* card (*i.e.* a proactive card) calls a service on a *server* card, it then continues the execution where it was previously stopped when the call occurred. To implement such a feature, our solution is based on the usage of the *switch* keyword where two cases are considered, *i.e.* before and after the invocation as shown in listing 3.

We have used the proactive mechanism with success to implement the collaborative services presented below and in the CBP – Air France application [1].

## 5.2. Distributed inter card services

A *distributed inter card service* is a service involving several cards. To perform its job, calls are done using the proactive mechanism presented above. Most of the time, the execution of the caller card depends on the execution of the receiver card.

Such a service could be a datamining service hosted on a master card, with the database and the entities performing

```

// b is a global variable previously initialized to 0

switch(b)
    case 0x00 :           // Common entry point: BEFORE
        instruction 1      // Instructions
        ...
        instruction n      // to do
        instruction n       // before to call the service

        call to the stub class // Format the request
        // and send it to execution environment

        increment b          // Save that a call is in progress

        break;                // Return from the Vlgg(process) method and
        // real emission of the call in the I/O interface

    case 0x01 :           // Entry point for the continuation of code: AFTER
        get the result        // Result of the request

        instruction n+1      // Instructions
        ...
        instruction n+m     // to do
        instruction n+m     // after the call to the service

        Set b to 0            // Save that the call is done

        break;                // Return from the Vlgg(process) method
        // and send the APDU response

```

Listing 3: Continuation for the proactive mechanism.

the search process: distributed over a set of other cards. When an external client calls the master card, this card in turn calls each card involved in the process so as to start the local datamining algorithm with the specified parameters, and then collects all the results before sending them back to the client. In this example, the local datamining service on each card means nothing in itself; it exists only because it is a part of a global computation. For this reason we can consider it as a microservice and not a service.

### **5.3. Composition of services**

Contrary to the distributed inter card service presented above, the composition of services enables to perform a complex task as an aggregation of services with possible dependencies. In this case, the call from a card to another card for a whole service are done in a transparent way (*i.e.* without the knowledge about if the called service is executed on the server card or distributed on several cards behind). This approach is a bit different than the previous inter card services where the service to achieve is distributed on several computational resources.

Such services are currently in development in our platform. Moreover we also plan to experiment about the composition of service at external client side with the University of Sfax.

## 6. CONCLUSION AND FUTURE WORK

The platform is operational and some services have already been tested. These services can be executed in a secure way, because they are deployed on a secure hardware (smart cards) and the communications between them are also secure, even though this could not be presented in details in this paper [10].

At the beginning, the Java Card Grid project was only intended as a proof of concept. We only planed to build a prototype platform. But now that it is operational, we have found a lot of interest in the university community, the official agencies and the industry. For instance we got the “most innovative technology award”, delivered by a committee comprising industry leaders, at e-Smart 2005[1]. We are now planning to set up a platform with 1000 cards to deploy applications that can handle real size data. We also plan to use the next generation cards that should be commercially available in 2007. They will provide 1 Gigabyte of memory, efficient processors, a full Java virtual machine and a TCP/IP stack. This will make it possible to experiment on a real size distributed environment.

## Thanks

Our project is supported by:

- Axalto, Gemplus and IBM BlueZ Secure Systems (for the cards);
- SCM Microsystems and SmartMount (for the readers);
- Sun microsystems (for the overall platform).

We also thank: Fujitsu, Giesecke&Devrient, Oberthur Card Systems and Sharp for the Java Card samples; David Corcoran and Ludovic Rousseau for their work on pcsc-lite and the CCID generic driver.

## REFERENCES

- [1] Eve Atallah, Serge Chaumette, Franck Darrigade, Achraf Karray, and Damien Sauveron. A Grid of Java Cards to Deal with Security Demanding Application Domains. In *Proceedings of e-Smart 2005*, Nice, France, September 2005.
- [2] Fran Berman, Anthony J.G. Hey, and Geoffrey Fox. *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley & Sons, 2003.
- [3] Alvin T.S. Chan, Florine Tse, Jianlong Cao, and Hong Va Leong. Enabling distributed corba access to smart card applications. *IEEE Internet Computing*, pages 27–36, May/June 2002.
- [4] Serge Chaumette, Achraf Karray, and Damien Sauveron. The Software Infrastructure of a Security Platform Java Card based for the Distributed Applications. In *Submitted*, 2006.
- [5] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
- [6] GlobalPlatform. GlobalPlatform. <http://www.globalplatform.org/>.
- [7] William Grosso. *Java RMI*. O'Reilly & Associates, 2002.
- [8] Scott Guthery and Mary Cronin. *Mobile Application Development with SMS and the SIM Toolkit*. McGraw-Hill Professional, 2001.
- [9] Thimothy M. Jurgensen and Scott B. Guthery. *Smart Cards: The Developer's Toolkit*. Prentice Hall, 2002.
- [10] Achraf Karray. Calcul sécurisé sur grille de cartes à puce. Master's thesis, ENIS – University of Sfax, 2004.
- [11] Roger Kehr, Michael Rohs, and Harald Vogt. Mobile code as an enabling technology for service-oriented smartcard middleware. In *International Symposium on Distributed Objects and Applications*, pages 119–130, September 2000.
- [12] Inc Sun Microsystem. *Java Card 2.2 Runtime Environment (JCER) Specification*, 2002. Remote Method Invocation Service, chapter 8, pages 53–68.
- [13] Object Management Group. The OMG's CORBA Website. <http://www.corba.org>.
- [14] Karsten Ohme. Open source GlobalPlatform library. <http://sourceforge.net/projects/globalplatform>.
- [15] PC/SC Workgroup. PC/SC Workgroup Home. <http://www.pcscworkgroup.com/>.
- [16] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook 2nd edition*. John Wiley & Sons, 2000.
- [17] Christoph Siegelin, Laurent Castillo, and Ulrich Finger. Smart cards: distributed computing with \$5 Devices. *Parallel Processing Letters*, 11(1):57–64, 2001.
- [18] The Community Resource for Jini Technology. <http://www.jini.org/>.