The Software Infrastructure of a Java Card Based Security Platform for Distributed Applications

Serge Chaumette, Achraf Karray[⋆] and Damien Sauveron[⋆⋆]

LaBRI, Laboratoire Bordelais de Recherche en Informatique
UMR CNRS 5800 – Université Bordeaux 1
351 cours de la Libération, 33405 Talence CEDEX, FRANCE.

Abstract. The work presented in this paper is part of the Java Card™[1] Grid project[2] 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: smart cards. We do not pretend that the resulting platform can compare to a real grid in terms of computational power, but it serves as a proof of concept for what a grid with secure processors could be and could do. As of writing, the hardware platform comprises 32 card readers and two PCs to manage them. The applications that we run on our platform are applications that require a high level of confidentiality regarding their own binary code, the input data that they handle, and the results that they produce. Even though we know that we cannot expect our grid to achieve high speed computation, we believe that it is a good testbed to experiment on the security features that one would require in a real grid environment. This paper focuses on the software infrastructure that we have set up to manage the platform and on the framework that we have designed and implemented to develop real applications on top of it.

1 Introduction

The technologies that have recently emerged and that are now widely used, contribute to anonymize and virtualize the resources of the network (Java, .NET, Solaris containers - previously N1 Grid Containers -, User-mode Linux, etc.) and the way they are connected (WiFi, Bluetooth, etc.). This leads to a situation where computing resources can effectively be shared. Sharing resources means federating them and allowing potentially unknown people to execute their applications on the resulting platform. The notion of Grid [1,2] is a well known example of this kind of approach that is now supplied by

[⋆] LaBRI (Bordeaux, FRANCE) and University of Sfax, ENIS, TUNISIA

[⋆⋆] XLIM, UMR CNRS 6172 (Limoges, FRANCE)

[1] 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.

[2] The Java Card Grid received the best innovative technology award at e-Smart2005.

both universities and private companies. Of course, users of such systems must accept to have their applications executed on resources that are under the control of someone else who they potentially do not even know.

Security is then 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 applications executed on the same platform. Second, the 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 where it has been loaded to work out what it is doing, or even to trace the instructions executed by my processor. If I upload a code to the machine of someone else, nothing will prevent my code from doing some malicious operation (*e.g.* my application can take advantage of hardware errors [3]), even though sand box approaches can solve some of the problems.

Smart cards [4] provide solutions for these problems, 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. Furthermore, 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. The processors that can be found in standard workstations do not offer the same protections. At the software level, the cards and the applications that they embed are evaluated and certified by well defined procedures (*e.g.* ITSEC [5] - Information Technology Security Evaluation Criteria - or CC [6] - Common Criteria -) in government approved agencies or companies (*e.g.* ITSEF - Information Technology Security Evaluation Facility).

It should be noted that even though smart cards are not very usable to achieve effective computations right now, the resources that they provide in terms of memory and computing power [7] 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 designing secure real size grids. This platform and more precisely its supporting software framework is the topic of this paper.

The rest of this paper is organized as follows. In section 2 we present what the role of smart cards could be in distributed systems. We explain the difficulties, the pros and the cons of their integration and usage in such environments. In section 3 we describe the overall Java Card Grid platform at both hardware and software levels. Section 4 focuses on the communication stack of the framework which is one of the key points of the environment. In sections 5, 6 and 7, we overview the solutions that we have set up to overcome the other main difficulties that we encountered. In section 8 we present the most relevant related work and compare it to our own approach. We eventually discuss in section 9 the future evolutions of our platform.

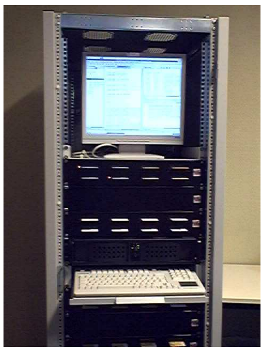
Fig. 1. The Java Card Grid platform.

2 Smart Cards in Distributed Systems

2.1 The Pros of Using Smart Cards in a Distributed System

Smart cards have a small size that makes them very mobile, *i.e.* it is easy to carry them in everyday life. It is a highly secure device that is able to store confidential information. It is protected against any attempt to obtain its contents. Contrary to most other devices, a smart card is resistant to physical attacks: it is a tamper resistant device. This is its main advantage. In fact, several mechanisms are integrated in a smart card at both hardware and software levels, so as to protect it against external attacks and thus protect the assets which it contains. In the worst case, the hardware structure of the card makes so that all the sensitive data (*e.g.* secret information or private keys) that it stores will be automatically destroyed in order not to be revealed to an attacker.

Smart cards offer a secure environment perfectly adapted to support higher level security features for large distributed systems. Nevertheless, their integration into such systems is not straightforward.

2.2 The Difficulties of Using Smart Cards in a Distributed System

There are many problems that prevent smart cards from being widely adopted in distributed systems.

First, smart cards are by design dependent on a device called the Card Acceptance Device (CAD), generally known as card reader.

Second, smart cards have very limited resources in terms of memory and storage capacity, computing power, and communication bandwidth with the outside. These limitations reduce the range of possible domains of application.

Third, 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 to execute, 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.

Fourth, the communication protocol between a card and its CAD is extremely poor, both in terms of contents and structure, and it is also very dependent on the brand of card. The communication takes place as a sequence of bytes (*i.e.* the APDUs – Application Protocol Data Units) which are poorly structured. The elaboration of the APDU-based messages must follow the ISO7816-4 standard [8] that describes their format and a number of special cases (error codes, special commands, etc.). Thus, the programmer must have a deep knowledge of the APDU protocol to be able to develop smart card applications. Compared to other communication methods such as RPC [9], RMI [10] or CORBA [11] which are widespread in distributed systems, the APDU protocol remains rather primitive and relatively complex, even though it is possible to use additional layers on top of it to partially hide this problem.

For all these reasons, the development of applications for smart cards is a complex task. It is really challenging to implement the mandatory layers that are missing if one wants to use smart cards in a distributed environment. We have developed these layers and they will be presented in sections 4 to 7.

2.3 Features Required to Support the Use of Smart Cards in a Distributed System

We have explained above why using smart cards in a distributed system is a complex process. Without a card reader and the proper software layers it is almost impossible, even though frameworks such as PC/SC can manage low level communication between the embedded applications and the host. To make it a reasonable task to develop applications that make use of smart cards, two main features have to be supported: high level communication and transparent service discovery. To make possible to develop real (size) applications, two other main features are required: proactivity and extended memory resources.

Communication. To face the complexity of the communication protocol and to make it easier so support different brands of cards, it is interesting to apply the methods of distributed systems to smart cards in order to provide transparent communication with the embedded objects. This is, for instance, what is offered by Java Card RMI [12].

Service Discovery. A smart card does not spontaneously announce the services that it contains. It is thus interesting to provide a mean for a card to expose and describe its services, so that the other components of the distributed system can discover and use them. This would make the integration of smart cards in distributed systems easier and faster.

Proactivity. 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.

Extended memory resources. The limited memory of smart cards is probably the main obstacle to the deployement of real size applications. Therefore, while cards with a large memory are not available, providing a solution to extend their capabilities is fundamental to experiment security in new kinds of application domains.

We offer these four features in the Java Card Grid platform. They are described in section 4 and section 5.

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 software.

3.1 Hardware Platform

The hardware platform that we have set up is presented Fig. 2. This figure shows that

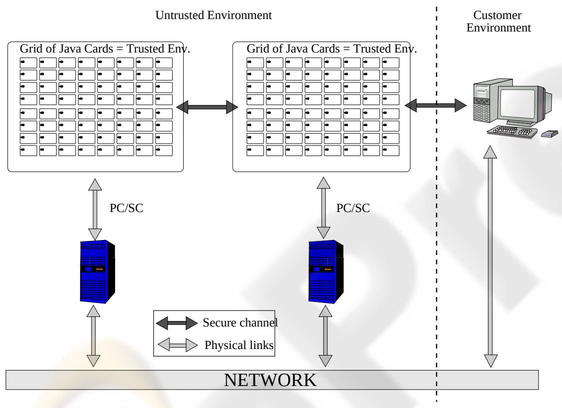

Fig. 2. A hardware platform based on Java Card grids.

we have in fact deployed two grids that are connected together by the network. 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 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 touchpad) that we use to control the servers. A picture of this platform is shown Fig. 1.

3.2 Software Framework

The software framework that we have designed and implemented comprises two layers: a low level layer 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.

At low level, the pilot PCs run Linux and we use `pcsc-lite` [13], an open source implementation of the PC/SC[3] standard, to manage the readers. We have chosen CCID[4] readers because they are supported by an open source generic driver which is available for `pcsc-lite` [15]. Because our high level framework is Java-based, we have added JPC/SC [16], a JNI-wrapper for PC/SC, to control the readers and to manage the applications embedded in the cards (*cf.* Fig. 3).

At the user level, the programming API is based on Mandala[17]. Mandala is a general framework that has been developed in our team to support distributed computing in Java. It provides a RMI-like abstraction with an *asynchronous* method invocation mechanism [18], which is very useful to deal with slow computing resources.

The overall software framework deployed on the two hosts is shown Fig. 3.

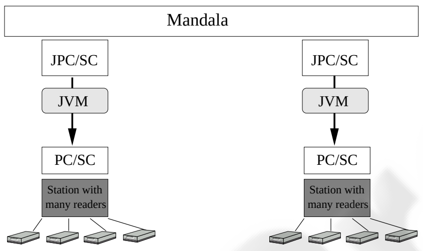

Fig. 3. The software framework.

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, only the remote topology control tool is available.

Remote Topology Control Tool. This tool enables a distant administrator to display the topology of the grid on its monitor, *i.e.* to see which readers are free and which contain a card. The administrator can also use it to track the evolution of the grid since any modification of its state, resulting from the insertion or the tearing of a card, is instantly notified to the tool. He can also get additional information about a given card and its associated reader: the low level communication protocol that it uses, the name of the reader, the name of the card, its ATR[5], etc.

[3] PC/SC [14] is a standard that provides a high level API to communicate with smart card readers.

[4] CCID is a standard that defines a protocol to handle USB readers.

[5] Answer To Reset.

Tools Under Development. The tools and APIs that we are currently working on are dedicated to the deployment of applets on a set of Java Cards. Since most of the Java Cards are GlobalPlatform[6] [19] compliant, we 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 [20] recently developed by Karsten Ohme.

4 Global Communication Stack of the Framework

We have used a number of different technologies when we have developed our framework and altogether they make a complex stack of software components and protocols. The goal of this section is to describe this stack and to explain the task of each layer. For the sake of clarity, we present it in terms of the OSI reference model to show the features of each software component and the complexity of the system. We eventually explain why future cards that support TCP/IP will make it possible for us to build a more flexible framework.

4.1 Current Communication Stack

Our framework is designed to handle several client applications (potentially running on different hosts) which interact with several Java Cards of the grid (more precisely with several applets possibly distributed on different Java Cards). These interactions take place through different servers and card readers. The servers of the grid and the readers act as gateways and there are only two really active entities in the dialog: the client application and the applet. All the software stack (including server processes) between these two elements needs to be transparent. In what follows, we present the architecture of this stack from the bottom to the top, by comparison with the OSI reference model.

Physical layer. At the bottom of the OSI model, the *physical layer* encompasses the physical interfaces and the physical protocols (*i.e.* voltages, timing, etc.). Between the client host and the server host we assume that the hardware interfaces are Ethernet cards. To simplify the stack presented Fig. 4, we also assume that the client and the server are in a LAN. In the real world, they could be in different networks and there might be several routers between them (but the framework would remain almost the same). On the grid side, the host machine and the CCID readers have a USB interface and the I/O channels use the USB protocol. The cards and the CCID readers are connected through contact interfaces (*i.e.* the contacts in the reader and the faceplate on the card) using the related physical protocol (*i.e.* the ISO7816 specifications). This layer is quite simple but yet requires three different protocols.

Data link layer. On top of the physical layer sits the *data link layer*, the goal of which is to provide error-free communication between network entities. It detects and possibly corrects errors that may occur in the physical layer. The client and the grid server both have an Ethernet module that communicates with the Ethernet cards. On the grid side,

[6] GlobalPlatform is a standard that specifies APIs to manage multiapplication cards.

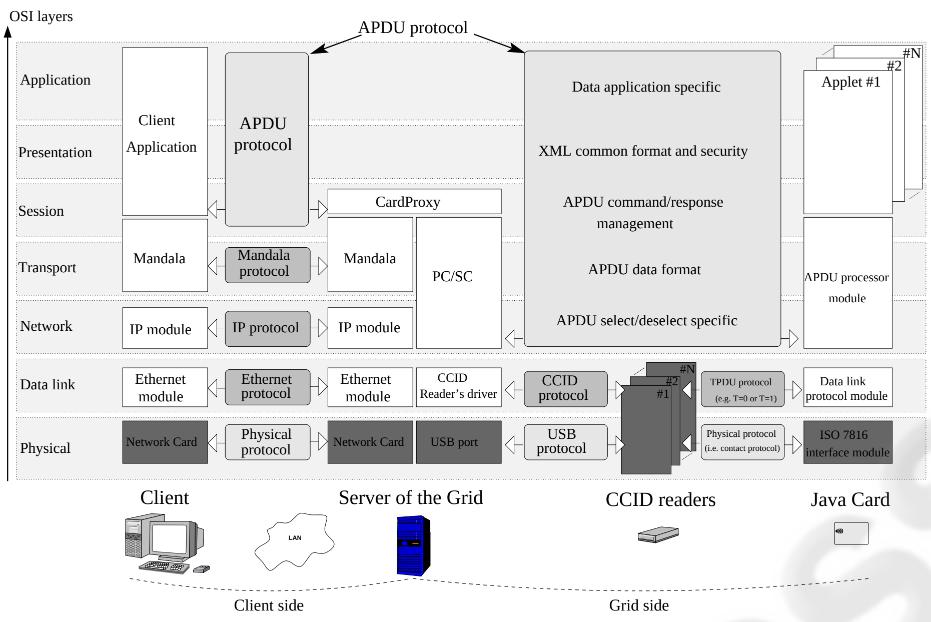

Fig. 4. Communication stack.

the data link layer is implemented by the CCID protocol between the CCID drivers hosted on the server and the CCID readers. Between the CCID readers and the data link module of the Java Card different kinds of TPDU (Transmission Protocol Data Unit) based protocols can be used (*e.g.* T=0 or T=1 are the more frequently encountered). The TPDU are constructed within the CCID readers and they are passed across the physical layer to the Java Cards. Thus, there are several different data link protocols between the different entities from the client application to the Java Cards.

Network layer. Over the data link layer, the OSI reference model has the *network layer*. One of its main tasks is to perform network routing. In the framework that we propose, there are several routing nodes. First, routing is achieved using the IP protocol, between the client and the server. As mentioned above, in a large network, many routers may be present between these two entities. On the grid side, the first routing element is the PC/SC component. It dispatches the messages that it receives to the specified CCID reader (or to the specified slot if the reader is a multi-slot reader) and then to the specified Java Card. The messages sent by PC/SC are called APDUs (Application Protocol Data Units) and there is no other protocol specified on top of it (at least for the transport layer and the session layer). Since this protocol encompasses many features that could be found at different levels of the OSI reference model, we decided, for the sake of clarity, to cut this protocol into five different subsets (where each subset corresponds to a part of the APDU specification) that we then map onto the five upper layers of the OSI model (*cf.* Fig. 4). Moreover, in our framework, the APDU protocol is not inside the same layers on the client side and on the grid side. We could not find a better solution with an architecture based on PC/SC. The APDU subset present at the level of the network layer is dedicated to the selection/deselection of applets in a Java Card. The next routing element is the APDU processor module that interprets the APDU selection/deselection commands and sets the target applet as the current active one, *i.e.* this applet will receive the following APDU commands (except if it is an other selection/deselection APDU).

Transport Layer. On top of the network layer, the *transport layer* provides connection services to the session layer. Between the client and the grid server our framework uses Mandala that supports several protocols to perform this task. One can observe Fig. 4 that Mandala is spread between the transport and the session layers; this will be explained a bit further in this paper. On the grid side, the protocol that is used is a subset of the APDU protocol representing the APDU data format specification. It sits between PC/SC on the server and the APDU processor module in the card.

Session Layer. Above the transport layer, we have the *session layer* that manages the dialog between the client application and the selected applet. In the OSI reference model, it provides either duplex or half-duplex mode of operation and establishes check-pointing, adjournment, termination, and restart procedures. In our framework, the client application and the CardProxy located on the server communicate using the APDU protocol. Thus, the application generates all the APDU commands to send to the distant applet and parses the APDU responses. The CardProxy acts as a gateway between the network communication technologies (*e.g.* IP) on the client side and the smart card communication technologies (*e.g.* CCID and TPDU) on the grid side (see Fig. 4). So, the upper layer enables direct end-to-end communication between a client application and an applet. The APDU protocol which is half duplex requires the use of a token to decide which entity can use the communication channel. Mandala also participates in the session layer by offering asynchronous communication features. These features can be used by both the client application and the CardProxy to achieve non blocking invocation of services, (at both client and server sides). On the grid side, the CardProxy and the applet also exchange data using the APDU protocol. The part of the APDU protocol specification mainly involved in this layer is the client/server model that schedules the commands and the responses. As soon as the CardProxy gets a response to a previous APDU command it checks its pending queue and if an APDU command is present, then it forwards it to the applet. The CardProxy also acts as a sort of router within the grid when a card wishes to call a method located on another card. It handles specific response messages to transform the server model of a smart card to a model where the card can also be a client. Thus it enables the card to be proactive (see section 6).

Presentation Layer. Over the session layer, the *presentation layer* relieves the application layer of concerns regarding syntactical differences in data representation within the end-user systems. In usual systems, MIME encoding, encryption and operations of that kind that have to do with the presentation of data are done at this level. In a similar manner, in our framework, the security and the data independent representations are managed in this layer. If this is required by the security policies, the communications between the client application and the applet can be ciphered. We also offer a XML based format to structure the data exchanged between the entities (*e.g.* to get the list of the available services and to pass the parameters when invoking a service on a card – see section 5). These two features are encompassed in a subset of the APDU protocol that we have defined, as presented Fig. 4.

Application Layer. The last layer on top of the stack is the *application layer* that is the direct interface of the application processes and that performs common services for them. Thus, in this layer, the data exchanged are specific to the goal of the application. Note that in our framework the application on the client side and the applet on the grid side each spread on three layers for practical reasons. However it would be possible to develop independent libraries on each side to handle the specific aspects of each layer. Nevertheless, it is important to be aware of the problems caused by the limitation of the resources available on Java Cards to achieve such developments (*e.g.* small stack of frames, etc.).

The framework and the stack of protocols that we have developed provide many of the features required by the OSI reference model. It was thus interesting to present our stack by comparison with the OSI stack even though it is not a direct mapping.

4.2 Future Communication Stack

In the short term we will make the communication stack much simpler by using TCP/IP Java Cards. These next generation cards will provide TCP/IP connectivity, what will make them directly accessible on the network. They will use a USB connection that will suppress the need for a reader and that will support high speed communication. This is very interesting in the case of our grid where the main cost is that of the readers. Moreover, thanks to the TCP/IP connectivity, we will have a homogeneous stack on the client side and on the grid side: TCP/IP for both. In such a context, the server of the grid will act as a simple router between the client applications and the applets.

These next generation Java Cards will also support a communication model in which the cards will be active, *i.e.* be able to initiate communication. This feature will be very helpful to call from inside a card, a service located in another card of the grid or on a distant host.

These two features, *i.e.* a TCP/IP stack and proactivity, will simplify our framework and open the way to new kinds of applications. Moreover these cards will be multi-threaded and this will make our framework really more efficient. The new protocols integrated in the Java Card specifications (*e.g.* TLS), the new APIs (*e.g.* cryptographic algorithms) and the new programming models will also make the development of new secure applications much simpler.

Nevertheless, to be able to work till these new cards become available on the market, we have proposed and implemented several mechanisms to use the current Java Cards.

5 Services in the Java Card Grid

By using the features provided by the stack presented above it is possible to easily and quickly develop 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 in a Service Descriptor using XML. This is then uploaded to the card and the service is registered in an on-card local registry, called *ServiceCatalog*, which contains all the Service Descriptors of the installed services. 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. More details about these different phases are shown Fig. 5 and are detailed in [21].

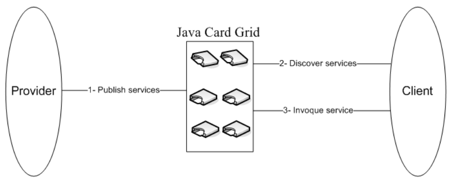

Fig. 5. Main steps to create, discover and invoque a service.

6 Proactivity in the Java Card Grid

To make the card active, *i.e.* able to send a request and thus, for example, to invoque a service on another card, we have set up a simple mechanism which consists in asking the card in order to know if it wishes to send a request. To achieve this goal, an APDU command is sent 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 in cellular phones [22,23].

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 of another card. Using 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, and the name of card where it is installed. More details about this extension are available in [21].

Fig. 6 illustrates the remote invocation method used between cards.

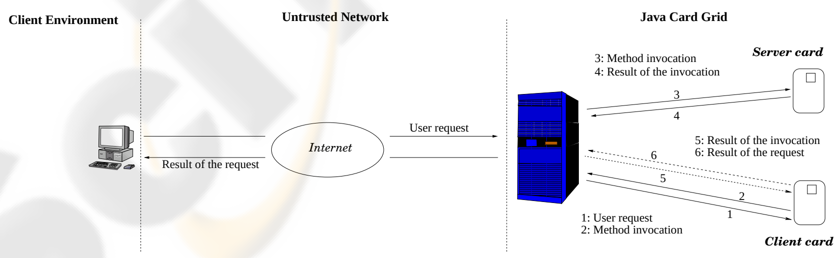

Fig. 6. Remote method invocation between cards.

7 Extended Memory in the Java Card Grid

In order to overcome the problems due to the small memory size of smart cards, we propose a solution that offers them a secondary storage (the hard disk of a host) which increases the memory capacity for the embedded applications:

- for short lived data, in a way similar to a virtual memory mechanism, as can be found in most operating systems (such as Windows, Linux, etc.);
- for long lived data, as with secure data repositories.

The proposed solution furthermore ensures the security so swapped information, *i.e.* it preserves the integrity and the confidentiality of data so as to be in perfect adequacy with the security requirements which characterize smart cards. Two entities are involved to achieve this goal:

- one on the card, a Secure Storage Manager Unit (SSMU) ;
- one on the host which performs the following operations:
  - write the ciphered data sent by the SSMU with specific keys for each applet and for each record,
  - manage the identification of ciphered data records,
  - return back data to the card (on demand).

Fig. 7 focuses on the entity deployed on the smart card side, the SSMU. It helps the applications that use a large amount of data to store it outside the card. It is furthermore in charge of ensuring the confidentiality and the integrity of the swapped data. Using the SSMU will therefore provide a good security level even though some of the application data are stored off-card. More details about this extension are available in [24,25].

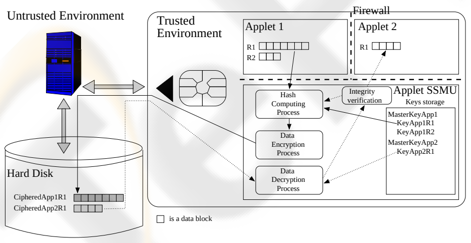

Fig. 7. Overview of the SSMU architecture.

8 Related Work

We have identified a number of projects, the aim of which is to integrate Java Cards in a distributed environment. The goal of all the frameworks that we have studied is to make the on card services usable as standard services: Corba[11] for ORBCard [26]; Jini[27] for JiniCard [28]; RMI[10] for JCRMI [12]. Once integrated, the functionalities offered on these cards 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 services. On the contrary, we intend to use the cards as cooperating computional resources [21]. To achieve this goal our framework makes the cards proactive and this is one of the major originalities of our work. Thus, in terms of services, our platform is more advanced and more flexible. Moreover, in our platform, both computation and inter card communication are secured. Finally, we have also set up a cache mechanism that is not available in any other smart card based platform.

9 Conclusion and Future Work

The platform that we have set up is working and we have run sample applications on it. We have solved the problems related to the physical equipments, the drivers and the communication layers required to make the whole system work. The security of our platform is based on the build-in hardware and software security mechanisms of the smart cards and on the security protocols used for the communication.

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 [29]. With one of the leading smart cards companies, we are now planning to set up a platform with 1000 cards to deploy applications that can handle real size problems. In the longer term, we also plan to use the next generation cards that should provide 1 Gigabyte of memory, efficient processors, a full Java virtual machine and a TCP/IP stack.

Acknowledgments

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. Berman, F., Hey, A.J., Fox, G.: Grid Computing: Making The Global Infrastructure a Reality. John Wiley & Sons (2003)
2. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (1998)
3. Govindavajhala, S., Appel, A.: Using Memory Errors to Attack a Virtual Machine. In: Proceedings of IEEE Symposium on Security and Privacy. (2003)
4. Rankl, W., Effing, W.: Smart Card Handbook 2nd edition. John Wiley & Sons (2000)

5. : ITSEC. (http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-fr.pdf)
6. : International Common Criteria home page. (http://www.commoncriteriaportal.org/)
7. Siegelin, C., Castillo, L., Finger, U.: Smart cards: distributed computing with $5 Devices. Parallel Processing Letters **11** (2001) 57–64
8. International Organization for Standardization: Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange. ISO (2005)
9. Sun Microsystems, I.: RFC1057 – RPC: Remote Procedure Call Protocol Specification. (http://www.ietf.org/rfc/rfc1057.txt)
10. Grosso, W.: Java RMI. O'Reilly & Associates (2002)
11. Object Management Group: The OMG's CORBA Website. (http://www.corba.org)
12. Microsystem, I.S.: Java Card 2.2 Runtime Environment (JCRE) Specification,. (2002) Remote Method Invocation Service, chapter 8, pages 53-68.
13. Corcoran, D., Rousseau, L., Sauveron, D.: pcsc-lite home page. (http://alioth.debian.org/projects/pcsclite/)
14. PC/SC Workgroup: PC/SC Workgroup Home. (http://www.pcscworkgroup.com/)
15. Rousseau, L.: CCID free software driver. (http://pcsclite.alioth.debian.org/ccid.html)
16. IBM BlueZ Secure Systems: Java Wrappers for PC/SC. (http://www.musclecard.com/middleware/files/jpcsc-0.8.0-src.zip)
17. Chaumette, S., Vignéras, P.: A framework for seamlessly making object oriented applications distributed. In: Parallel Computing 2003, Dresden, Germany (2003)
18. Vignéras, P., Grange, P.: The Mandala website. (http://mandala.sourceforge.net/)
19. GlobalPlatform: GlobalPlatform. (http://www.globalplatform.org/)
20. Ohme, K.: Open source GlobalPlatform library. (http://sourceforge.net/projects/globalplatform)
21. Chaumette, S., Karray, A., Sauveron, D.: Secure Collaborative and Distributed Services in the Java Card Grid Platform. In: Proceedings of Workshop on Collaboration and Security (COLSEC'06), Las Vegas, Nevada, USA (2006)
22. Jurgensen, T.M., Guthery, S.B.: Smart Cards: The Developer's Toolkit. Prentice Hall (2002)
23. Guthery, S., Cronin, M.: Mobile Application Development with SMS and the SIM Toolkit. McGraw-Hill Professional (2001)
24. Chaumette, S., Karray, A., Sauveron, D.: Secure Extended Memory for Java Cards. In: Proceedings of he 2006 International Conference on Computational Science and its Applications (ICCSA 2006), Glasgow, UK (2006) (Poster).
25. Chaumette, S., Karray, A., Sauveron, D.: Secure storage for the Java Card Grid. (In: (Submitted))
26. Chan, A.T., Tse, F., Cao, J., Leong, H.V.: Enabling distributed corba access to smart card applications. IEEE Internet Computing (2002) 27–36
27. The Community Resource for Jini Technology. (http://www.jini.org/)
28. Kehr, R., Rohs, M., Vogt, H.: Mobile code as an enabling technology for service-oriented smartcard middleware. In: Internationnal Symposium on Distributed Objects and Applications. (2000) 119–130
29. Atallah, E., Chaumette, S., Darrigade, F., Karray, A., Sauveron, D.: A Grid of Java Cards to Deal with Security Demanding Application Domains. In: Proceedings of e-Smart 2005, Nice, France (2005)