Secure distributed computing on a Java Card™* Grid

Serge Chaumette Pascal Grange Achraf Karray† Damien Sauveron‡
Pierre Vignéras
LaBRI, Laboratoire Bordelais de Recherche en Informatique
UMR 5800 – Université Bordeaux 1
351 cours de la Libération, 33405 Talence CEDEX, FRANCE.
{chaumett,grange,karray,sauveron,vigneras}@labri.fr, http://www.labri.fr/

Abstract

More and more pieces of hardware are being connected to the Internet every day. Technologies such as Bluetooth or Wi-Fi make this evolution even faster. To make these equipments cooperate and communicate with each other several paradigms such as mobile codes, mobile agents and remote procedure calls are particularly well adapted. These paradigms enable to execute a code that is either coming from somewhere over the network, or that is local but managed remotely. Security is then one of the main concerns that has to be dealt with. We believe that smart cards, and more precisely Java Cards can help to cope with this challenge. This is a position paper where we present the first results obtained on a Java Card based platform that we have set up for experimentation purpose. These experiments raise many questions we are currently working on.

1. Introduction

One of the reasons for setting up a grid [6, 15] or for connecting computing resources together is to allow people or companies to use computing units provided by third parties. The problem is that the user has to trust the owner of the computing resource where her code will be executed. Even though some security can be provided at the software level, nothing can prevent malicious operations that could result from physical access to the computation unit: confidential data can be eavesdropped, calculation can be disrupted and results can be tampered with.

We believe that the use of smart cards [17] can make it possible to cope with these problems. The physical protection that they offer ensures that it will be infeasible, in a reasonable amount of time, to understand what is stored inside them and what occurs internally. This is not the case with standard processors. Moreover, it is now acknowledged that the ever increasing computing power of smart cards should allow to achieve some effective calculation [22] even though it is clear that high performance cannot be expected. Therefore we are setting up a grid or cluster of Java Cards to experiment on these security problems. This is the topic of this paper. We believe that our work will enable to define a general approach to provide secure computing on third party hardware.

The rest of this paper is organized as follows. Section 2 gives an overview of related work. In section 3 we present the Java Card Grid project [9, 10]. Then in section 4 we describe the first demonstration implemented on a small cluster of Java Cards. Next we explain the choices that we have made to bypass the problems encountered to set up the demonstration. This will help in the process of designing a bigger cluster of Java Cards and implementing a programming framework on top of it. In section 6 we present the benchmarks that we have run on our demonstration Java Card grid. The last section is dedicated to work in progress and we describe an additional application that will take advantage of the security features of our framework.

2. Related work

Frameworks such as JiniCard [19], Jason [7] or OrbCard [8] have been developed to communicate in transparent and secure ways with the services offered by smart cards. Although these frameworks do not directly consider distributed computing with smart cards as their target paradigm, it is still possible to use them to achieve this goal. However they miss features that are useful in the context of distributed applications, such as asynchronous method invo-

---

∗ 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. All other marks are the property of their respective owners.

† University of Sfax, ENIS, TUNISIA

‡ LaBRI (Bordeaux, FRANCE) and LMSI (Limoges, FRANCE).

cation [14, 18] that is especially required when slow computing resources (like smart cards) are used.

More formal approaches to protect mobile codes executed on untrusted runtime environments have been developed. For instance, there is an original but typical solution based on an extension of function hiding using error correcting codes [20]. These solutions, although supported by strong foundations, seem to be difficult to use in practice because of the assumptions they make on the applications.

3. The Java Card Grid project

Based on the multi-applicative feature of the Java Card technology [13, 23] and on our experience of both this technology and distributed computing, we believe that it is possible to set up a cluster of smart cards and to provide a software framework for developing and managing secure applications on this cluster.

We understand a *software framework* as the APIs for the developer and the tools for the end-user or the administrator. This framework will be based on some pre-existing system developed for distributed computing such as RMI [16], JavaParty [21], JiniCard [19], Jason [7], OrbCard [8] or Mandala [12]. Mandala is a general framework that has been developed in our team to support distributed computing. It provides a RMI-like abstraction. Mandala offers features that we believe are useful in the context of this work such as the *active container* [11] concept it is based on, or the *asynchronism* it provides for remote method invocation [26].

The hardware platform that we are targeting is presented Fig. 1. The smart cards all together make up some sort of grid or more precisely of cluster. They are powered by the smart card readers themselves. These USB readers are chained together and connected to a certain number of hosts which will be used to manage them.

4. The Demonstration

We have built a prototype platform and developed a demonstration application to validate the feasibility of our project before developing a real size platform and the associated framework. In this section, we give a detailed description of the Mandelbrot demonstration application. We do so because we then use it both to elaborate on the challenges encountered to set up our platform and to run benchmarks.

4.1. Overview

Our demonstration application computes the Mandelbrot set using the DJFractal project [25]. DJFractal is yet another fractal generator which uses Mandala [26] to distribute the computation over a set of CPUs ranging from workstations to Java Cards.

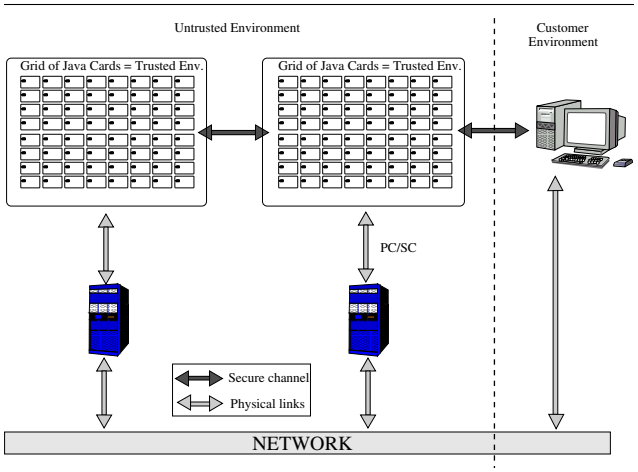

Figure 1. A solution for computing with Java Cards

DJFractal uses several entities in order to achieve the computation. First, it defines a *fractal data*, fd, as a rectangular subregion $fd = (x, y, width, height)$ of the whole area to be computed – with some additional information. Next, it defines a *fractal computer* as an object able to deliver a *fractal result* from a given fractal data. The initial area is split into several fractal data (see section 4.3) that must be given to one of the possibly many fractal computers. Though not required, there usually exists only one fractal computer per host. As of writing, the number of fractal computers is stable during the computation (new devices and crashes have no influence). A *scheduler* is used to decide which fractal computer must compute the next fractal data. Hence, the scheduler must know the whole set of available fractal computers. This is achieved using a collection of fractal computers in the sense of the package `java.util.Collection`[1].

In order to increase performance, we concurrently compute several fractal data. Based on the Mandala framework that allows any public method of any object to be invoked asynchronously and remotely, DJFractal uses asynchronous method invocation on both local fractal computers and remote ones.

1 We might have used the `java.util.Set` class which better represents the mathematical notion of set (collection with no duplicate element), but this imposes many constraints such as the impossibility to use a `java.util.ArrayList` as a set.

4.2. Prerequisite: Mandelbrot Set Definition

We consider the complex plane $\mathcal{P}$, where

$$\forall (x,y) \in \mathbb{R}^2, P(x,y) \in \mathcal{P} \equiv z = x + yi, z \in \mathbb{C}$$

For a given $z \in \mathbb{C}$, we consider the sequence $M(z)$:

$$M(z) = \begin{cases} z_0 = z, \\ z_{n+1} = z_n^2 + z \quad n \in \mathbb{N} \end{cases}$$

The *Mandelbrot set* is the set of $z \in \mathbb{C}$ for which $|M(z)|$ remains bounded. It can be proven that for any given z for which $|z| \leq 2$, if there exists a *count* n such that in $M(z)$, $|z_n| \geq 2$ then $|M(z)| \rightarrow \infty$ and z is outside the Mandelbrot set. This criterion may seem not too valuable, as it only works when $|z| \leq 2$. However, it is known that the entire Mandelbrot set lies inside the disk centered at the origin and of radius 2, so these are the only values of z that we need to consider.

4.3. The Algorithm

The method used by DJFractal is inspired by the algorithm described by Peter Alfeld [5]. This algorithm computes (and draws as soon as possible) the interesting structure of the Mandelbrot set. Informally, the interesting structure is the set of points lying at the border of the Mandelbrot set.

For every point P in the complex plane, DJFractal must determine if it lies inside the Mandelbrot set or not. For this purpose, the algorithm computes the *count* n of points in $M(z)$. Note that if z is in the Mandelbrot set, it means that no *count* has been found for which $|z_n| \geq 2$. So a *bailout* $B \in \mathbb{N}$ is defined at the beginning of the computation so that if $|z_B| < 2$ we assume that z is in the Mandelbrot set. So the $count(z)$ function is defined as:

$$count(z) = \begin{cases} n, & \text{if } \forall i < n, |z_i| < 2 \text{ and } z_n \geq 2 \\ B, & \text{if } \forall i \leq B, |z_i| < 2 \end{cases}$$

In order to draw the set, we assign to each pixel $P(x,y) \equiv z = x + yi$ in the complex plane a color $color(z)$ which depends on $count(z)$. In particular, if $count(z) = B$, then $color(z)$ is set to black.

For the computation of a given fractal data fd, *i.e.* a rectangular set of pixels, the algorithm distinguishes two cases depending on $size(fd)$:

- if $size(fd) > S$, where S is a constant known before the computation, then fd is split into four new fractal data. A measure of the importance of each region regarding the structure of the Mandelbrot set must then be determined using their *discrepancy* value, δ, defined as follows :

$$\delta(fd) = max(K_{fd}) - min(K_{fd})$$

$$K_{fd} = \{count(lt_{fd}), count(lb_{fd}), count(rt_{fd}), count(rb_{fd})\}$$

where $lt_{fd}, lb_{fd}, rt_{fd}$ and rb_{fd} respectively stand for the left-top, left-bottom, right-top and right-bottom corner of the fractal data fd. As an example, consider a computation where $B = 100$, and a fractal data where $K_{fd} = \{10, 65, 78, 100\}$. Then $\delta(fd) = 100 - 10 = 90$. The fact that the discrepancy is so high suggests that the fractal data contains a significant structure.

After the computation of $\delta(fd)$, each pixel of the fd region is given a color $c(fd)$:

$$c(fd) = color(average(K_{fd}))$$

and the subregion is drawn on the screen.

- else, $size(fd) \leq S$, and each pixel z of the region is sequentially assigned its color $color(z)$;

The algorithm always takes fractal data from the top of the highest discrepancy stack – there is one stack per discrepancy value –, queries the scheduler to work out which fractal computer these data must be sent to, and depending on the fractal data size, asynchronously invokes (and remotely if the fractal computer is remote) the method `discrepancy()` or `calculateAll()` as illustrated Fig. 2. It starts with the whole region on the stack. Doing so the algorithm always refines a fractal data with the highest currently represented discrepancy. Fractal data with low, e.g. zero, discrepancies, for example those in the interior of the Mandelbrot set, are left till the end.

If t is the number of instructions necessary to compute one iteration of $M(z)$, then since $count(z) \leq B$, the complexity for a fractal data is:

$$C(fd) \leq t.B.size(fd)$$

4.4. The scheduler

The DJFractal application can use various types of schedulers to take advantage of the underlying hardware/software infrastructure. The complete description of this framework and the detailed description of the various schedulers available is out of the scope of this paper. Therefore we will only briefly describe the scheduler used for the benchmarks presented in section 6.

The algorithm we use to compute the Mandelbrot set is designed so that it is possible to perform the computations in parallel and to use several computers. To take advantage of this possibility, we need a scheduler that chooses, for

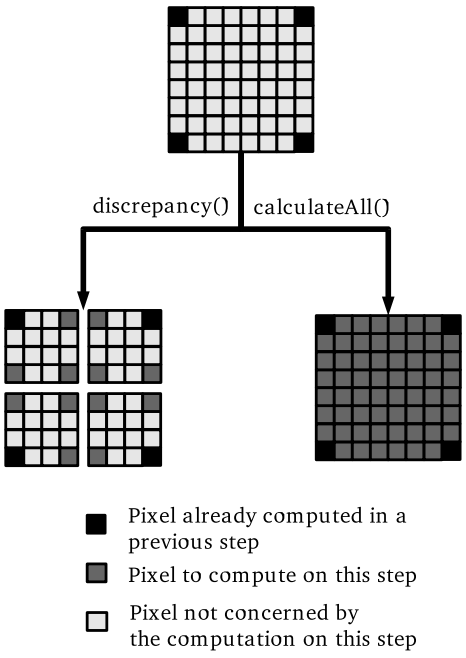

Figure 2. An illustration of the algorithm used in DJFractal.

each task to achieve, the best suited computer. Of course, we used Java Cards for the particular case of this demonstration[2] and all these processing units had the same performances. However, we did not rely on this assumption of homogeneous performances since we can use heterogeneous Java Cards.

Our scheduler has to choose a processing unit for each computation to perform. It is dedicated to the computation of the Mandelbrot set and we do not plan to reuse it in another context. The main characteristics of the algorithm used to compute the Mandelbrot set are the unpredictable irregularity of the computation and the fact that current and future work is generated by previous work.

Two types of operations are involved in the calculation of the Mandelbrot set: `discrepancy()` and `calculateAll()`. As seen in section 4.3, `discrepancy()` requires the computation of three points and generates four new computations to perform later. `calculateAll()`, however, requires the computation of many more points, that is all the points of the corresponding area. This is a first source of irregularity. Moreover, the amount of computation needed to compute the value of one point is totally unpredictable. Therefore we decided to implement a simple scheduler based on fifo lengths: each fractal computer has an associated fifo where its pending requests are stored. From the hypothesis that a fractal computer which is more efficient than others or which was submitted easier computations will have a smaller amount of pending requests than others, we submit the current computation to the fractal computer with the smallest number of pending requests. We used this scheduler in the benchmarks presented in section 6. It significantly improved the performances of the overall computation compared to round robin scheduling.

The main advantage of this solution is that it only relies on the size of the pending queues to choose among the fractal computers. It is also its main drawback. When all the fractal computers have the same amount of pending requests, no differences appear and the scheduling falls down to round robin. If during a given period of time all the fractal computers are involved in a computation, then, during that period, the scheduler will send future computations to the fractal computers with smaller pending queues. After a certain amount of time, all the pending queues will have the same size. We call t the interval of time between that instant and the instant when some fractal computer terminates its current computation and reduces the length of its queue. During t, the scheduler submits the future computations in a round robin way. Suppose that one of the fractal computers is significantly slower than the others. Then during this interval t, it will get the same number of computations to perform as the others even if it exhibits poor performances. This fractal computer can perform slower because it uses slower hardware or because of the complexity of its pending computations that may be significantly harder to compute than those submitted to the others. The poor performances of Java Cards led to the multiplication of the period of time where all the computers are busy, *i.e.* all pending queues have the same length, and led us to reconsider our scheduler.

We introduced the notion of bounded pending queues. We decide on a maximum size for the pending queues. When all the fractal computers have their pending queues full we just stop submitting jobs. That way, we avoid performing round robin scheduling. When some fractal computer reduces the size of its pending queue, we submit unsubmitted tasks still relying on the lengths of the pending queues.

Another interesting characteristic of this algorithm is the starvation issue. At the end of the computation, some fractal computers may be idle while some are performing computations and have non-empty pending queues. In that case it would be interesting to consider the pending requests for resubmission to the idle fractal computers. The algorithm

2 Note that the framework we use is not specific to Java Cards.

we use to compute the Mandelbrot set may exhibit starvation[3]. Since work is generated by work, there may be situations where there is no more task to perform before some `discrepancy()` has been computed. Several fractal computers are idle while others have non empty pending queues and idle fractal computers have to wait till new computations are generated. The bounded scheduler limits this problem but it is not sufficient and we are currently working on this issue.

4.5. Technical Platform

The hardware of our demonstration platform is made up of 2 PCs, 9 USB CCID readers, 2 USB hubs with 7 slots and 9 Java Cards. We are in the process of deploying additional hardware.

Figure 3. The Java Card grid.

The PCs run on a Linux environment. Our framework is Java-based therefore we use JPC/SC [1], a JNI-wrapper for PC/SC to control the readers and the deployment of the applications (*cf.* Fig. 4). PC/SC [4] is a standard which provides a high level API to communicate with smart card readers. We chose JPC/SC rather than OCF [2] because it is very easy to use and we have deep knowledge of PC/SC since we contribute to the development of the `pcsc-lite` project [3]. Moreover OCF is not maintained any more and cannot handle as many readers as PC/SC. A solution based on OCF, PC/SC and a bridge from OCF to PC/SC would make it possible to handle as many kinds of readers as PC/SC but we did not want to add an overhead by using additional software layers. Each PC/SC compatible reader

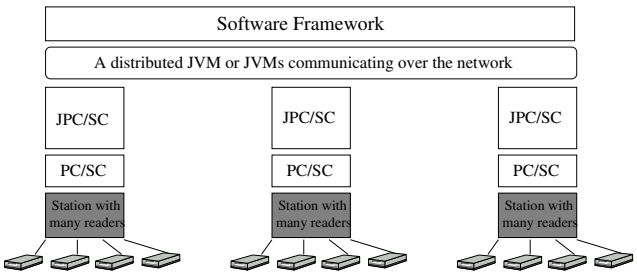

Figure 4. The software framework

comes with a pluggable driver used to communicate with the PC/SC middleware (*cf.* Fig. 5). Using only PC/SC com-

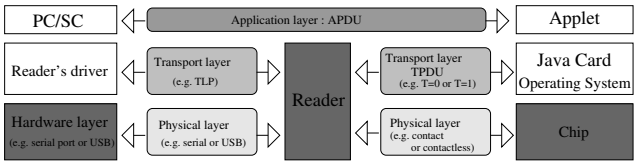

Figure 5. The PC/SC solution

patible readers makes it possible to use readers of different suppliers without deep knowledge of the underlying specific protocols.

For the purpose of our demonstration application, we implemented Java Card applets able to compute `discrepancy()` and `calculateAll()` for a given fractal data. On the host computer, a proxy is implemented that translates incoming calls to fractal computers into APDUs[4] to the target applet.

5. Challenges

5.1. Solved challenges

To set up this demonstration we have solved a number of challenges:

- since the type double is not available in Java Card, we have implemented a class Double. There are many traps to implement such a class. For instance the call stack and the memory are limited and the objects allocated persist during all the life of the card.
- some readers use closed proprietary drivers and thus we decided to only consider readers with open source drivers to cope with implementation problems. We only use CCID readers because this standard allows

3 This case trivially appears at the beginning since only one area exists and it leads to the creation of four new areas to compute.

4 APDU is the elementary message to communicate with smart cards at the application level.

various CCID [24] readers from different manufacturers to be supported by the same driver.

- we have added new features to `pcsc-lite`:
  – the support of 255 readers at the same time;
  – a new improved management of the requests sent by the clients to the readers that enables to make the calls in parallel;
  – the support of the new version of the `pcsc-lite` driver interface;
  – some security controls such as the verification that only the thread owning a PC/SC context can release it.

5.2. Remaining challenges

We have identified the following challenges that we will have to cope with in order to achieve secure computing on our cluster of Java Cards:

- Memory size. To handle this constraint we could for instance cipher and store the intermediate results in a standard (unsecure) memory with large capacity and fast access. We could also use a virtual memory distributed over a number of dedicated smart cards, what would make it possible to keep the data secret, but we still would have to cipher the data to transfer them securely through the insecure channels between the cards.
- Insecure channels. The communication has to be ciphered between the clients and the grid of smart cards, and between the smart cards within the grid itself. We have begun to set up such a framework.
- Execution model. To be efficient, our framework has to offer asynchronism and an active mode of operation allowing a smart card to behave as a server or as a client. We are working on a method to call code on a card from the host and from another smart card.
- Heterogeneity and deployment of applications. To be usable our framework has to be transparent, to manage the heterogeneity of the hardware (smart cards, readers and hosts), and to provide fault-tolerance mechanisms. It is also necessary to manage the deployment, the termination and the mapping of the pieces of code over the cards.

6. Benchmarks

We have run our demonstration several times, each time using 9 cards from a different manufacturer. We have also run it on a mixed set[5] of cards. Fig. 6 is a screen shot of what is displayed when the demonstration is running.

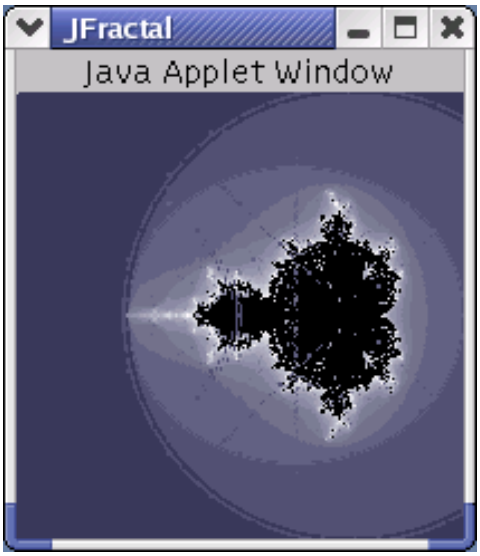

Figure 6. Screen shot of the Mandelbrot Fractal computed over the cards.

The resulting measures are shown in TAB. 1. The initial fractal data is a 200x200 pixels area defined by its top-left and bottom-right corners in the complex plane $(-3,-2);(1,2)$. The parameters are $B = 20, S = 100$ and the scheduler is as described in section 4.4.

Model	Execution time (min.)
9 Java Card onto Fujitsu mb94r215b (FRAM)	122
9 GemXpresso Pro R3 E32PK	202
9 JCOP31bio	253
9 GemXpresso Pro R3 E64PK	376
9 SmartC@fé Expert	389
Mixed	344

Table 1. Benchmark without secure channel.

Fig. 7 shows some tests of scalability. Since we do not have a sequential version of the algorithm available, the comparison is to the time of our parallel algorithm running on one single Java Card. The shape of the curve is not surprising (although when using 6 cards we could expect better performance – there was certainly a problem that we did not identify, such as a dead card): the speed up is initially

5 3 GemXpresso Pro R3 E32PK, 2 JCOP31bio, 2 GemXpresso Pro R3 E64PK and 2 SmartC@fé Expert.

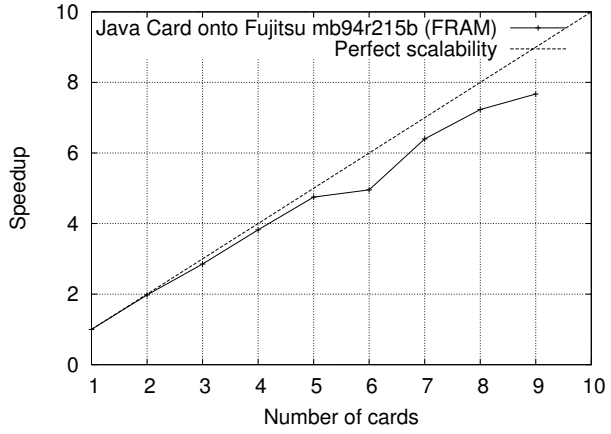

Figure 7. Tests without secure channel.

very good, and it reaches a threshold when the number of cards is increasing and then begins to decrease.

Although we did not yet measure the overhead due to our framework we performed some tests with a secure channel and the overhead is negligible. Indeed, computing a DES on the chips that we have tested is approximately $100\mu s$ because it uses a hardware crypto-processor. Considering that the secure channel that we have to implement between the entities of our platform consists in a 3DES for all the APDU commands (from the reader to the card) and that the number of messages is limited, then the overhead should be very small.

Of course this is clearly not high performance computing but the feature we are focusing on is security, not speed. In addition the results presented here do not mean that the slowest cards should not be used: they may be more secure. Indeed the speed often decreases while the security increases.

The good results of the Fujitsu chip is mainly due to the 32 bits architecture and the technology of the non-volatile memory (NVM), the FRAM being faster regarding write operations than EEPROM. According to an analysis of our applet provided by Giesecke&Devrient our code mainly tests the memory access to the NVM. These benchmarks have thus confirmed that using cards based on EEPROM NVM should be avoided. Indeed this kind of memory supports only 10^5 write cycles whereas the FRAM suppports a minimum of 10^{12} write cycles. In the future, as far as possible, we will use Java Cards with memory based on FRAM or Flash.

7. Work in progress, demonstration application

We are currently working on an additional application that will take advantage of the security features of our framework. The aim of this application is to show how we can ensure the confidentiality of both the data and the code of a given application, each being the property of two separate entities that do not trust each other.

The first entity is the FBI that is willing to analyze the passengers files for Air France flights. Of course, the FBI does not want its analysis algorithm to be known by Air France. The second entity is Air France that is willing to cooperate, but does not want (and this has recently been enforced by the EEC) personal confidential data about its passengers to be delivered to any external organization.

To deal with these constraints we are implementing an application on our platform (in fact a first basic prototype is already working). First, we distribute the passengers file over a set of Java Cards. Each passenger has been given a key that makes it possible for Air France to identify her and that is used by external entities, making it possible to hide the effective names of the passengers. Each card is provided with a light API - supposedly developed by Air France - so that another applet loaded on the card can access the non confidential part of the passengers data. Second, the code of the FBI is also distributed over the cards and uses the API provided by Air France to analyze the passengers file. When a given criteria is met, the FBI application comes back with a key that represents the suspicious passenger. It can then get back to Air France and ask for more information about this given passenger, information that Air France is then free to provide or not.

8. Conclusion

To implement the demonstrations presented in this paper we had to cope with a number of challenges. By doing so we gained experience on secure computing over a grid of Java Cards. We believe that we will be able to apply this experience to a bigger platform. We furthermore think that this first step will make it possible to securely handle any piece of equipment that can be connected to a network and that provides Java Card-like hardware level security.

Thanks

Our project is supported by:

- Gemplus (for the cards);
- 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 and Oberthur Card Systems for the Java Card samples. We also want to thank David Corcoran and Ludovic Rousseau for their work on `pcsc-lite` and the CCID generic driver.

References

[1] Java Wrappers for PC/SC. `http://www.musclecard.com/middleware/files/jpcsc-0.8.0-src.zip`.

[2] OpenCard Framework. `http://www.opencard.org/`.

[3] pcsc-lite home page. `http://alioth.debian.org/projects/pcsclite/`.

[4] PC/SC Workgroup. `http://www.pcscworkgroup.com/`.

[5] P. Alfeld. The Mandelbrot Set. `http://www.math.utah.edu/~alfeld/math/mandelbrot/mandelbrot.html`.

[6] F. Berman, A. J. Hey, and G. Fox. *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[7] R. Brinkman and J.-H. Hoepman. Secure Method Invocation in Jason. In *Proceedings of the 5th Smart Card Research and Advanced Application Conference (CARDIS'02)*, pages 29–40, San Jose, California, USA, 2002.

[8] A. T. Chan, F. Tse, J. Cao, and H. V. Leong. Enabling Distributed Corba Access to Smart Card Applications. In *IEEE Internet Computing*, pages 27–36, May/June 2002.

[9] S. Chaumette, P. Grange, D. Sauveron, and P. Vignéras. Computing with Java Cards. In *Proceedings of CCCT'03 and 9th ISAS'03*, Orlando, FL, USA, July 31, August 1-2 2003.

[10] S. Chaumette and D. Sauveron. The Smart Cards Grid Project. `http://www.labri.fr/Perso/~chaumett/recherche/cartesapuce/smartcardsgrid/documents/poster.pdf`. Poster presented at Cartes 2003.

[11] S. Chaumette and P. Vignéras. Active containers: an alternative approach to mobile agents systems. Proceedings of the Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98, Santa Fe, NM, USA. Poster.

[12] S. Chaumette and P. Vignéras. A framework for seamlessly making object oriented applications distributed. In *Parallel Computing 2003*, Dresden, Germany, September 2-5 2003.

[13] Z. Chen. *Java Card™ Technology for Smart Cards: Architecture and Programmer's Guide*. The Java™ Series. Addison-Wesley, 2000.

[14] D. Donsez, S. Jean, S. Lecomte, and O. Thomas. (A)synchronous Use of Smart Cards Services Using SOAP and JMS. In *Proceedings of the Gemplus Developer Conference 2001*, Paris, France, June 20-21 2001.

[15] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[16] W. Grosso. *Java RMI*. O'Reilly & Associates, 2002.

[17] *Smart Card Handbook 2nd edition*. John Wiley & Sons, 2000.

[18] S. Jean, D. Donsez, and S. Lecomte. Smart Card Integration in Distributed Information Systems: the Interactive Execution Model. In *Proceedings of IEEE International Symposium on Advanced Distributed Systems (ISADS'2000)*, Guadalajara Jalisco, Mexico, march 2000.

[19] R. Kehr, M. Rohs, and H. Vogt. Issues in Smarcard Middleware. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of Lecture Notes in Computer Science, pages 90–97. Springer-Verlag, 2000.

[20] S. Loureiro and R. Molva. Mobile Code Protection with Smarcards. In *Proceedings of the 6th ECOOP Workshop on Mobile Object Systems*, Cannes, France, June 2000.

[21] M. Philippsen and M. Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, November 1997.

[22] C. Siegelin, L. Castillo, and U. Finger. Smart cards: distributed computing with $5 Devices. In 2001, editor, *Parallel Processing Letters*, volume 11, pages 57–64.

[23] Sun microsystems. *Java Card™ 2.2.1 Specifications*. Sun microsystems, 2003.

[24] USB Implementers Forum. Universal Serial Bus Device Class Specification for USB Chip/Smart Card Interface Devices version 1.00. `http://www.usb.org/developers/devclass_docs/ccid_classspec_1_00a.pdf`, march 2001.

[25] P. Vignéras and P. Grange. The DJFractal project. `http://djfractal.sf.net/`.

[26] P. Vignéras and P. Grange. The Mandala website. `http://mandala.sourceforge.net/`.