An evaluation of the Java Card environment

Christophe Rippert[*], Daniel Hagimont[†]

Contact: Christophe Rippert, Sirac Laboratory
INRIA Rhône-Alpes, 655 avenue de l'Europe
Montbonnot 38334 St Ismier Cedex, France
Phone: +33476615501, Fax: +33476615252
Email: Christophe.Rippert@inria.fr

Advanced Topic Workshop: Middleware for Mobile Computing
November 16, 2001
Heidelberg, Germany
http://www.cs.arizona.edu/mmc/

Abstract

In this paper, we present the experiment we have conducted to evaluate the suitability of Java cards for advanced applications. We developed a resource-consuming application in a Java card using only the specified Java Card environment, and noted which missing features could have been useful to improve the functionalities of our application or to ease its development. We then propose new system services to improve the Java Card environment.

1 Introduction

The advent of the Java Card technology [1] in 1996 opened new perspectives for smart card application providers. However, the use of Java cards has mostly been reduced to small applications, like electronic wallets or ID cards, for which their potential does not appear to be completely used. It therefore seems interesting to try and evaluate the Java Card environment for programming bigger applications. Since hardware limitations are bound to disappear as manufacturing techniques improve, we only focus on software limitations in this paper. We developed a resource-consuming application and noted the limitations of the Java Card environment which hindered the development. This experiment enabled us to identify new system systems for the Java Card environment which might be needed to ease the development of advanced applications.

We first present the Java Card environment. Then we describe our application and point out the limitations of the Java Card environment which caused us problems during the development of the application. Finally, we propose new system services to improve the Java Card environment and address the problems previously identified.

2 The Java Card environment

A Java card is essentially an Integrated Circuit Card (ICC) with an embedded Java Virtual Machine. The Central Processing Unit (CPU) can access three different types of memory: a

[*]Université Joseph Fourier (Joseph Fourier University), Grenoble, France
[†]Institut National de Recherche en Informatique et en Automatique (French National Institute for Research in Computer Science and Control), Grenoble, France

persistent read-only memory (ROM) which usually contains a basic operating system and the greatest part of the Java runtime environment, a persistent read-write memory (EEPROM) which can be used to store code or data even when the card is removed from the reader, and a volatile read-write memory (RAM) in which applications are executed.

Java Card applications are written using the Java Card language. This language is a subset of the Java language [2]. Noticeable restrictions concern the primitive types available which are limited to booleans, bytes and shorts, and arrays which must be linear. Since Java Card applets are compiled using a plain Java compiler, they must undertake an additional checking phase to verify that they comply with the previous restrictions. Due to limitations of the Java Card class loader, the class files must then be converted to a simpler binary format before being loaded on the card.

On the card, applets are executed in the Java Card Runtime Environment (JCRE) [3]. The JCRE includes the Java Card Virtual Machine (JCVM) [4], the Java Card Application Programming Interface (API) [5] classes and support services like the applet loader for example. Due to the small quantity of memory available on the card, the JCVM is significantly restricted compared to the Java Virtual Machine [6]. Main differences include no dynamic class loading, no garbage collection, and no support for multiple threads of control. The Java Card API is also very limited.

Java Card applets are composed of two parts: a server part on the card and a client part run on the terminal linked to the reader. This client/server scheme is based on a low level protocol of communication through message exchanging. Application Protocol Data Units (APDUs) are sent by the terminal to the card to issue commands and the card send back responses in the same way. It is important to note that the card can only send APDUs as responses to a client request and cannot call the terminal on its own initiative.

Before being usable, the server part of an applet must be loaded on the card. This operation is done by the applet installer, which is usually an applet dedicated to that use provided by the card manufacturer. Once loaded, an applet is initialised by calling the method `install` that all Java Card applets must implement. This method is usually used to allocate static fields and initialised them. It must finish by calling the method `register` which notifies the JCRE of the new installed applet. Since the card may contain several applications, the JCRE must maintain a list of all installed applets.

Once it has been initialised, the client can select an applet by sending a select APDU to the JCRE which in turn calls the method `select` of the applet. This method can be used to initialise data that must be initialised every time the applet is run (and not only once when it is installed). Then the client can send command APDUs to the JCRE which passes them to the method `process` of the applet. The method corresponding to the command APDU sent by the client is then called and possibly returns a result. This result can be sent back to the client in a response APDU. Finally, the client can select another applet by first calling the `deselect` method of the applet through a deselect APDU and then selecting the new applet.

3 Application example

To test the suitability of smart cards for real applications, we have developed a car driver's assistant application. This application becomes more and more common nowadays, even in cheap cars. The user simply enters his destination and guidance information is displayed on a screen and updated as the car moves. This application is based on three hardware components: the Java card on which the application runs, the terminal embedded in the car which is composed of a screen, a keyboard and of course a Java card reader, and the GPS which gives the current position of the car to the application. A wireless network connection can optionally be added, to allow dynamic downloading of maps for example.

Sons per node	Best path length					
	2	3	4	5	6	7
2	1.24	2.19	2.97	3.80	4.68	5.61
3	2.10	4.35	6.12	7.85	9.54	11.19
4	3.15	7.19	10.38	13.41	16.27	18.96

Figure 1: Computation times in the card (times are in *seconds*)

Sons per node	Best path length					
	2	3	4	5	6	7
2	0.55	0.57	0.59	0.61	0.64	0.67
3	0.66	0.70	0.72	0.75	0.77	0.82
4	0.77	0.81	0.84	0.87	0.90	0.94

Figure 2: Computation times on the PC (times are in *milliseconds*)

3.1 Execution scheme

The application provider sells a Java card which contains the application and the maps of several towns. The user inserts his card into the terminal and selects his destination from the proposed list. The map of the current town is explored (i.e. the application computes the best path from the current location to the appropriate town exit to get to the next town) and guidance information are displayed as the car moves. When the car reaches the town exit, the map of the next town is explored and so on until the user reaches his destination. Additional maps can be added after the card issuance, for example by downloading them from the application provider's web site into the card if the terminal include a wireless network connection.

We chose this application as an example since it is memory and CPU-time expensive. The maps are represented as graphs, which consume a significant amount of memory when representing large maps. Moreover, the map-solving algorithm is a simple width-first search of the graph, a rather slow and memory-consuming algorithm on a smart card. One must note though that this application is not meant to be realistic and that alternative architectures could be devised to achieve much higher performances. We purposely chose a resource-consuming architecture to push the Java Card environment to its limits, not a realistic example to illustrate its use.

3.2 Prototyping issues

We developed this application using Gemplus GemXpresso RAD 211 [7]. The Java card used includes an eight bits processor running at 5 MHz, 32 KB of ROM, 32 KB of EEPROM, and 2 KB of RAM. The embedded Java Card environment is compliant with the Sun Java Card 2.1 Specifications.

The application is divided in two parts: the client part (on the terminal) which includes a graphic user interface, and the server part (in the Java card) which comprise the main application code. The client part (approximately 1300 lines) is written in standard Java 2 whereas the server part (around 540 lines) is subject to the restrictions of the Java Card language.

We performed some benchmarks to compare the performance of our application in the card and on a standard personal computer (i.e. a PC with an Intel Pentium II processor and 128 MB of RAM). Since the most time consuming part of the application is the search for the best path in a map, we choose to monitor that function on both platforms. The parameters the most interesting to make vary are the maximum number of sons per nodes and the length of the best path computed since the maps are represented as graphs. We give in Figure 1 and Figure 2 the results of these benchmarks for the function in the card and on the PC.

3.3 Prototyping constraints

The major problem we had to deal with was obviously the lack of memory in the card. The maps we used to test the application included approximately 20 nodes and the temporary data needed to find the best path in a town filled all the available memory. The algorithm used to find the optimal path is a simple breadth-first search of the tree representing the possible paths in the map, which requires an array to store the nodes of the same level during the search. Since the size of this array grows exponentially with the height of the tree, it soon becomes to big to fit in the card.

Moreover, since the JCRE does not include a garbage collector, all object allocations must be done at installation time (i.e. when the application is downloaded in the card) and not at run time. Since the objects are never unallocated, reallocating them each time the application is run would quickly fill the card memory. So we had to know when we developed the application the size of all the arrays we used. This was impossible for some, since their size depended on an user input. This constrained us to set the sizes of these arrays to their maximum possible value, which worsened the memory shortage problem described above.

Another problem we encountered concerns the lack of any dynamic remote class loading mechanism in the JCRE, which prevented us from improving the flexibility of our application. For example, if the application provider wants to modify the format of his maps, he must update the application embedded in the cards. The first way to do it would be to recall all the issued cards, which would be very expensive, and bothersome for users. Another way would be for the application to update itself automatically. For example, if the application connect to the application provider web site to download a map using the optional wireless network connection, it can detect that a new format is available, and download the code plug-in that will enable it to read that new format and translate it in its own internal format. To do that, a dynamic remote class loading mechanism would be necessary since the JCRE would have to download and link the code plug-in to the application. Unfortunately, this service is not available in the JCRE.

In our application, the terminal is only used for user interaction and as a gateway for the GPS. Since the main operations (i.e.: finding the best path, ...) are executed in the card, it would be logical to consider the card as the client to the terminal which would act only as an input/output device. Unfortunately, the Java Card communication model imposes that all commands come from the application on the terminal (i.e. the applet in the card can only response to commands and never issue them). This master/slave model is constraining for the design of an application since it forces a code architecture which might not be the one which the programmer would like to set up.

Finally, the communication scheme between the terminal and the card is not very comfortable to use. As we have seen above, it is based on APDUs exchange. Unfortunately, the maximum length of these APDUs (approximately 250 bytes) makes the passing of large arguments (e.g.: arrays, complex objects, ...) to a method on the card rather tedious since these arguments must be split by the client and then merged again on the card. Moreover, methods on the card cannot be called "directly" (i.e.: by using their symbolic name) and they must be assigned an identifier (an integer) which the client includes in the APDU to select the method it wants to call. It is then the task of the process method to decode the APDU and call the chosen method.

3.4 Synthesis

We can differentiate the hardware and software limitations detailed above. The memory size and CPU power limitations are bound to be suppressed as the card manufacturers improve their miniaturization techniques (recent cards now include as much as 2 MB of persistent memory). So our study focuses only on software limitations and we expect the following services to be realistic on recent hardware.

4 Missing services

In this section, we present the new system services that could be added to the JCRE to improve its features or ease of use. It should be noted that we were not able to actually implement any of these services on a Java Card since to our knowledge there does not exist any open-source Java Card Virtual Machine. Moreover, empty ICCs are not sold freely to prevent credit card piracy. Therefore, our propositions remain empirical, but we expect recent hardware progresses to cope with the additional cost induced by the implementation of these services in the JCVM.

4.1 Improved communications between the terminal and the card

4.1.1 Remote method invocation

Providing a remote method invocation service, like the one included in the Java 2 Platform, would greatly improve the programmer's comfort and efficiency, since he would not have to cope with APDUs which are uncomfortable to use. This service is sometime supplied by card manufacturers, but since it is not yet specified in the Java Card Platform, all these proprietary implementations are incompatible with one another. Examples of such proprietary implementations include Gemplus DMI [8].

4.1.2 Reversing the client/server scheme

With the communication protocol specified in the Java Card platform, the methods on the card are inherently passive: they can be called and return a result but they cannot call a method on the terminal on their own initiative. This scheme can be unsuited for application for which the main program should logically be located in the card. It might be interesting then to reverse this master/slave model. Allowing nested method invocations might also be interesting, though it might require support for multi-threading in the JCRE if the card is allowed to issue the first command.

4.2 Dynamic class loading

The standard Java 2 environment provides a mechanism which enables the Java Virtual Machine to download remote classes and link them dynamically when they are needed (this mechanism is used by the Java Remote Method Invocation [9] protocol for example). The JCRE does not provide such a service. The applet installer can load applets on the card and call their install method, but it cannot be invocated by another applet since it is usually an applet itself and to select it would mean deselecting the other applet, thus losing its execution stack. So this applet installer cannot be used to dynamically load classes like it can be done in the Java 2 Platform. Providing such a service could be interesting, for example to update applications as we have seen above.

4.3 Advanced memory system

The Java Card memory system is very restrictive compared to the Java 2 platform. As we have seen above, objects must be allocated at installation time since they cannot be unallocated. A garbage collector could be implemented to enable object unallocation. Since the JCVM is mono-threaded, this garbage collector would not be able to work will another applet is running, but it could be called in the `deselect` method for example. Thus, an applet could allocate objects dynamically during its execution and these objects would be unallocated when it terminates.

However, the main problem with Java cards remains the very small size of memory available. Technical limitations complicate the installation of memory, especially RAM, on small devices like smart cards. So it could be interesting to study a swapping system which would use the

memory installed on the terminal to discharge the memory on the card. Such a service is economically interesting since buying a few megabytes of RAM for a PC for example is very cheap compared to the cost of memory for smart cards. This service could be implemented in the operating system, in which case it would probably be similar to existing swapping systems in UNIX or Windows and would swap pages of memory. Or it could be a part of the JCVM which would enable it to swap whole objects. In that last case, a serialization service would be necessary to save and restore the state of swapped objects.

5 Conclusion

As we have tried and shown in this paper, the Java Card environment is still very limited compared to the standard Java 2 environment. The memory system limitations are very constraining for the programmer and the lack of a dynamic remote class loading mechanism forbids whole types of applications. Extending the Java Card environment with advanced system services is therefore necessary if it is to be used for realistic applications. The cost of these services remains a pending question, but recent progresses in hardware let us believe that their implementation should not be too penalizing in term of resource-consumption. By providing these extensions to their virtual machines, Java cards manufacturers would enable advanced applications to be developed, thus finally exploiting fully the potential of smart cards.

References

[1] Java Card Technology home page. Sun Microsystems, Inc. `http://java.sun.com/products/javacard/index.html`.

[2] The Java Language Specification, Second Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. Addison-Wesley, June 2000.

[3] Java Card 2.1 Runtime Environment (JCRE) Specification. Sun Microsystems, Inc. February 1999.

[4] Java Card 2.1 Virtual Machine Specification. Sun Microsystems, Inc. March 1999.

[5] Java Card 2.1 Application Programming Interface. Sun Microsystems, Inc. February 1999.

[6] The Java Virtual Machine Specification, Second Edition. Tim Lindholm, Frank Yellin. Addison-Wesley, April 1999.

[7] GemXpresso 211 product sheet. Gemplus. `http://www.gemplus.fr/developers/products/gemxpresso_rad211/index.htm`.

[8] Direct Method Invocation. Jean-Jacques Vandewalle, Gemplus. `http://sirac.inrialpes.fr/ecole/99/cours/99-13.pdf`.

[9] Java Remote Method Invocation Specification. Sun Microsystems. December 1999. `http://java.sun.com/products/jdk/rmi`.