Improving Java Card Grid Dependability with Fault Prevention and Fault Tolerance

Monia Ben Brahim, Maher Ben Jemaa, and Mohamed Jmaiel
ReDCAD Laboratory
National School of Engineers of Sfax
BPW 1173 Sfax Tunisia
Emails: monia.benbrahim@redcad.org Maher.benjemaa@enis.rnu.tn Mohamed.Jmaiel@enis.rnu.tn

Abstract— **We present in this paper a solution that increases the dependability of a Java Card Grid platform by improving the grid service availability and continuity. Our solution is based on fault prevention and fault tolerance. It mainly tolerates material faults, communication faults, and faults related to the easy mobility of a smart card. Fault tolerance assures that the grid provides the appropriate answer to the client despite the failure of the active card service. It is implemented by applying a passive replication strategy at card services level as well as by the possibility to substitute the failed card service by a composition of other card services. Fault prevention permits to prevent the use of an unavailable card service in order to avoid the need for a recovery. The evaluation of execution times shows that the overhead of the added software layer is relatively acceptable either when there is no fault or when the recovery is made by replication. However, the overhead is much bigger when the recovery is made by composition.**

I. Introduction

An inherent problem of distributed systems is their dependability [4], [16]. Indeed, the dependability is the property that allows system users to grant a justified confidence to the delivered service. It answers to several requirements such as the reliability, the availability, the maintainability, and the security of the system.

The Java Card Grid platform (JCG) [3], [7], [10] is a distributed platform that is particularly secured since it is based on the Java Card technology [12], [17]. However, it doesn't consider the handling of faults related to cards and their readers such as a dysfunction of the card or its reader, or their extraction in inattentive or malicious way. These faults, when they occur at execution time, affect the grid state and lead to the omission failure of the JCG platform that becomes unable to answer to the client's request. Even when they occur while cards are inactive, they may affect the service availability in the grid.

The JCG platform dependability was seen only from the security angle [3], [7], [9], [10], [15] and in our knowledge no prior work treated reliability and availability issues in a JCG. In this paper, we present a solution that increases the dependability of a JCG platform by improving the grid service availability and continuity. Our solution is based on fault prevention and fault tolerance. It takes the fault handling into account by preventing the reuse of a failed service. It also handles the error by assuring that the grid provides the appropriate answer to the client despite the failure of the active card service. The error handling is performed through applying a passive replication strategy at card services level and through extending this strategy with the possibility to substitute the failed card service by a composition of other card services. The recovery by composition is possible when the abundance and the diversity of services, held in the grid, allow composition of card services having varied complexity degrees. We preserve of the passive replication strategy the aspects of replication and passivity of replicas. As card services we deal with in this work are stateless, it becomes useless to synchronize the duplicates. The possibility to substitute a failed card service by a composition of services permits to tolerate the fault even if all secondary replicas are unavailable (failed, retired of the grid, or already active). It also allows, if the duplication is only intensified at elementary card services level, to favor the recovery by composition of services.

The fault tolerance layer that we designed and implemented includes three software components: the recovery manager, the global service registry, and the message monitor. The interaction between these components is transparent to the user and assures a fault tolerant execution of his application on the JCG platform. Also, it was designed in such a way that permits to deal with the possible faults that can occur during the handling of a previous error. In addition, we improved the global service registry architecture in order to prevent the use of an unavailable card service. We implemented some prevention mechanisms such as monitoring of events and access in mutual exclusion.

Our evaluation tests show that our approach minimizes the execution time overhead when there is no fault, and keeps a proportionally acceptable execution time compared to the initial one when the recovery is based on replication. However, the recovery by composition, although it is not expensive from the resources replication perspective, seems to be costly from the execution time perspective.

This paper is organized as follows. We describe in section 2 the JCG platform that we deployed. In section 3, we enumerate the faults that we tolerate in the JCG and present the fault tolerance layer. In section 4, we present the faults to prevent and the different prevention procedures. Then, section 5 discusses the evaluation results of our approach. Section 6 presents next the related works and the necessary background. We finally conclude and present perspectives of our work.

II. The Java Card Grid Platform

We set up a Java Card Grid [3], [7], [10] at small scale constituted from a set of Java cards (in our case 4 Java cards) and card readers. A photo of the grid is shown in figure 1 and the corresponding hardware architecture is presented in figure 2. The software infrastructure is made of two layers: a low-level layer that manages communications between clients and cards. The second layer, can be considered high-level, has service oriented architecture and includes phases of publication, discovery, invocation, and composition of services. A card service represents an applet installed in the card and contains methods that can be invoked through APDU[1] commands. Every card service has a descriptor so that it can be published by a service provider, discovered and invoked by a distant client. The service descriptor [10] is an XML file describing the interface of the applet that implements the service. We also developed the necessary tools for the composition of card services on client side: a description language that is XML-based and dedicated for card service composition, the orchestration engine that interprets this language, and a graphic tool that facilitates the discovery, the invocation and the composition of services in the JCG platform [5].

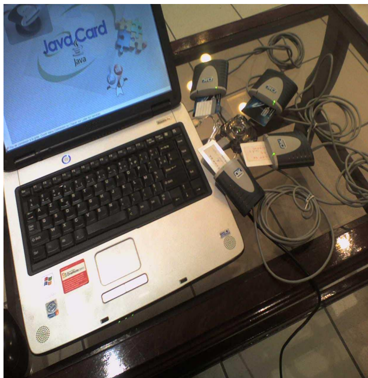

Fig. 1. The Java Card Grid platform.

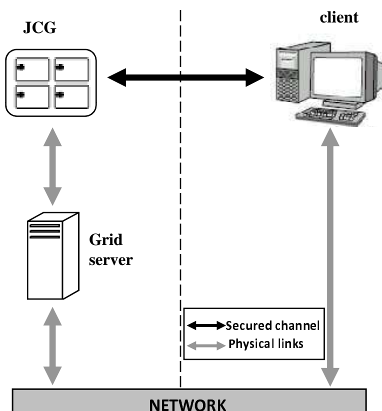

Fig. 2. The JCG hardware architecture.

[1]Application Protocol Data Unit, protocol of communication with the card.

III. Fault tolerance in the JCG

In our approach, error handling can be either by replication or by card service composition.

The recovery by replication is a JCG recovery in which the occurred error is handled using a passive replication. A replica corresponds to the couple: card service, smart card. The secondary service may not be an identical copy of the primary card service. In fact, the java card grid platform has service oriented architecture with many possible service providers.

The recovery by composition is a JCG recovery in which the occurred error is handled by substituting the failed service by a composition of other card services. This composition is semantically equivalent to the failed service.

The recovery manager, the global service registry, and the message monitor are the software components we developed in order to tolerate faults occurring in the JCG.

A. *Faults to tolerate in the JCG*

The faults that we tolerate in the JCG platform are mainly material faults, communication faults, and faults related to the easy mobility of a smart card. Material faults include any dysfunction of the card or its reader. This dysfunction can be, for example, the result of some problems in the electrical alimentation of the card (or the reader). We also distinguish the case where the card is saturated, that means it reaches the maximal number of writing cycles on the card memory.The smart card then becomes useless. Communication faults occur when there is not any more connection with the card. We are interested in the case where the connection between the card reader and the grid server is failed.

The smart card portability and mobility motivate us to think about faults related to the removal of a card (or its reader) inattentively or even intentionally.

B. *The recovery manager*

The recovery manager is composed of two main entities: the recovery engine and the support modules. The recovery engine executes the recovery logic. Available recovery actions are published via the manager interface.

According to the architecture shown in figure 3, the recovery engine does not perform the recovery actions directly but, instead, uses a set of support modules. This implies that its recovery capabilities depend on the functionalities provided by the used modules. The advantage of this solution is that the single recovery modules can be updated, added or removed easily and without having to implement the whole recovery engine every time.

The cryptographic protocol used for securing the communication between the client and the grid [15] requires the use of session keys that must be shared only between the client and the card (the grid server is not considered as a trusted entity). Thus, only the client entity can construct secured requests and send them to grid cards. In order to maintain secure execution, the recovery manager is implemented on the client side.

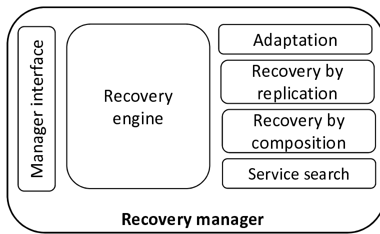

Fig. 3. The architecture of the recovery engine.

1) Recovery by replication: We consider that each active service (i.e. service that is invoked by a client) in the JCG is a primary service. Other equivalent services are secondary services (even if they are activated by other clients). When the primary replica is failed, this module allows its substitution by a secondary replica. The failed APDU command will be sent back to the new primary service. The remaining commands, destined to the failed service, will be redirected to the new card service.

2) Recovery by composition: This module permits to substitute the failed service by a composition of card services. It exploits the orchestration engine [5] developed for the platform composition of services layer. Having the composition description as well as the descriptors and localizations of services involved in this composition, the recovery by composition module launches the orchestration engine to generate the appropriate APDU commands and sends them to the grid. The composition output is then encapsulated in an APDU answer that will be returned to the calling application. The remaining commands, destined to the failed service, will be also substituted by the same composition of services.

3) Adaptation: Sometimes the primary and the secondary services don't have exactly the same description interface. This could imply that the recovery by replication module is not able to communicate with the new service because it cannot recognize its messages. The adaptation module hides the differences between the two descriptors by transforming the exchanged messages.

4) Service search: This module provides operations to dynamically discover a secondary service or a composition of services when an active service is failed. Service compositions are previously described [5] and stocked in the grid server. In order to facilitate the service search functionality, we implemented, on the server side, a global service registry containing all necessary information about available card services in the grid as well as their possible compositions.

C. The global service registry

A special Java Card applet must be present in every card in the JCG. It represents a local service registry that contains descriptors of all services installed in the card. Since it is possible in the grid that a card moves from a reader to another, a service descriptor cannot contain any information about the localization of this service. The secondary replicas cannot be therefore defined in a static way. Having a global service registry (see figure 4) gathering all the descriptors of available services in the grid and managing their localizations and their semantic equivalences (duplications and compositions) facilitates and accelerates the search of secondary replicas and service compositions for the recovery.

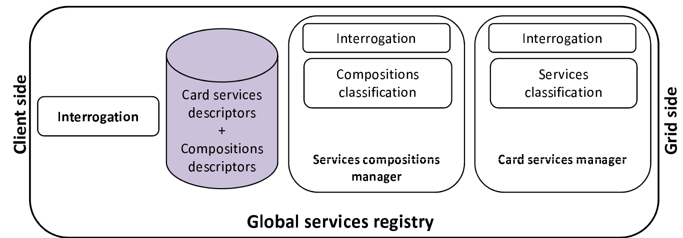

Fig. 4. The architecture of the global registry of services.

1) Service classification: Service classification consists in gathering the equivalent services in only one group. In the same way equivalent service compositions are gathered in only one group since several different compositions can lead to the same result. At the server starting, the card service manager explores the grid in order to catch descriptors of all available services and to detect the localization of every service. Then, the classification module of this manager creates, using configuration files or ontologies [18], the corresponding service groups.
Moreover, the classification module of the service compositions manager discovers all compositions description files stocked in the server and creates the corresponding composition groups.

2) Interrogation of the global service registry: The interrogation module allows querying the global service registry about the existence of a secondary card service or a composition of services that can substitute a failed service. The priority, while searching, is for secondary card services.
Sometimes, a service included in a found composition is not available. In this case, it is necessary to look for, if it exists, another composition; otherwise to look for another service (or another composition) semantically equivalent to the unavailable service. The priority is granted to the composition implicating the smallest number of services.

D. The message monitor

Error detection is essential to trigger the recovery process. During the communication with a card service, any failed APDU command provokes the generation of a server level exception. It is the *CardProxy* that raises this exception when the connection with the card becomes impossible and no APDU answer is gotten. We chose, in our JCG platform, the *JPCSC*[2] *API* to communicate with a Java card. This API uses the *PCSCException* class for *PCSC* [1] related errors. Every exception has a symbolic representation expressing its message, reason code plus symbolic representation of reason code. The following table shows two examples of these symbolic representations. We then added to the software layer,

[2]A JNI (Java Native Interface) wrapper for PC/SC

TABLE I
EXAMPLES OF SYMBOLIC REPRESENTATIONS OF AN EXCEPTION IN THE *JPCSC API*

Fault	symbolic representation of the exception
Card removal during a communication	*SCardTransmit(): 0x8010002f, General error*
Reader removal during a communication with its inserted card	*SCardTransmit(): 0x1f, General error*

responsible for fault tolerance in the platform, a module that supervises the communication between the client application and the grid. This module that we call *message monitor* detects and recognizes any exception related to a fault occurring during the communication with a Java card. The *message monitor* must afterwards send back the exception toward the client application in order to start the recovery process.

E. Interaction between Fault tolerance layer entities

If any service invocation is fault tolerant in the JCG, the user application involving several card services is necessarily fault tolerant too. When a fault is happening, the different fault tolerance layer entities must interact all together in order to detect the error and to ensure a correct service invocation. We detail this interaction while commenting figure 5.

Interaction 1: before sending a request to the server, the invocation layer keeps the request historic (APDU command, name and localization of the card service) in the recovery manager.

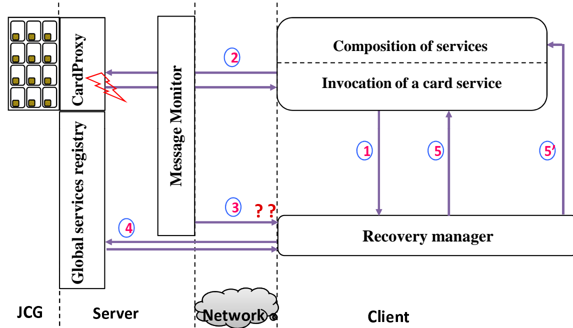

Fig. 5. Fault tolerant service invocation.

Interaction 2: the CardProxy sends, in turn, the request to the considered service and waits for an APDU answer. The Java card holding the service is then active in the grid and if no fault occurs, the card service sends back its APDU answer to the CardProxy which relays it to the invocation layer.

Interaction 3: if, during the communication, a fault occurs and no APDU answer is returned, the CardProxy raises an exception. The message monitor recognizes this exception and notifies the problem to the recovery manager in order to trigger the recovery process.

Interaction 4: the recovery engine starts recovery process execution by interrogating the global service registry about the existence of a secondary service or a composition of services to substitute the failed service.

Interaction 5: when the global service registry answer requires a recovery by replication, the recovery manager sends back the APDU command that has just been memorized toward the secondary replica. Interactions 1 and 2 are then going to be repeated. We emphasis this recursion in order to treat faults that can eventually occur during the communication with the secondary service. So we can have recursive recoveries when a fault occurs during the handling of a previous error.

Interaction 5': if a recovery by composition is required, the recovery manager executes the composition of services using the JCG platform composition layer. Since the orchestration engine executes the involved invocations through the card service invocation layer, the handling of faults occurring during the composition execution is always assured.

IV. FAULT PREVENTION

A. Faults to prevent

The main fault that we aim to prevent is the use of a failed or unavailable card service. Indeed, the user can inattentively invoke a non-existent service in the grid, i.e. the card holding this service is removed from the grid either during a previous communication with it or while it was inactive.

The user can also choose to invoke, without knowing, an existent but failed service, for example when the card holding is dead.

The simultaneous access of users to different services installed in the same card is also a fault that could lead to an inconsistent service invocation. In fact, the virtual Java Card machine doesn't support Java threads [17]. So at a time, only one applet can be executed in the virtual machine.

All these faults can be also committed by the fault tolerance layer during the use of service (s) for the recovery process. Besides, this layer can make mistakes related to descriptions files of the service compositions such as using a description file that is not saved anymore in the server.

B. Prevention procedures

In order to avoid the signaled faults, we characterized every card service object recorded in the global service registry by a state that can be ”available” or ”unavailable”, and we improved the registry architecture by adding modules controlling the existence of services in the grid as well as their states (see figure 6).

A service having the ”unavailable” state is a service that exists in the grid but it is active or failed. The client entity (respectively the global registry interrogation module) must verify the service state before choosing it for an invocation (respectively for a recovery).

1) Events monitoring in the JCG platform: The updating module of the card service manager listens, through the *”EvntListener”* module, any event occurring at the grid level. Then, the events listener gives this information to the *”EvntAnalyser”* module that does the appropriate actions to each kind of event.

Particularly, if a card is removed from its reader or a reader containing a card is removed, the events analyzer deletes, from the global registry, all services objects associated to the

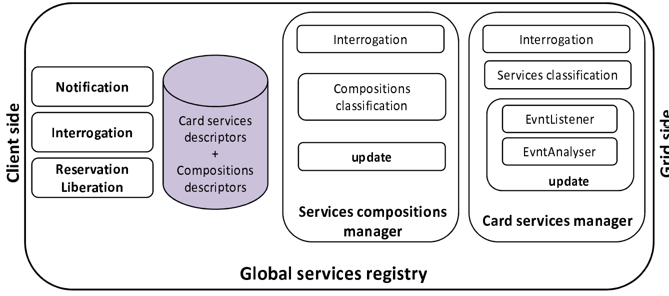

Fig. 6. Improved architecture of the global service registry.

services held in this card. Next, it signals to the notification module to inform the client applications about the occurred event. The service list displayed on users' side is then updated in order to avoid that users invoke a card service already removed from the grid.
The updating module of the service compositions manager also controls any suppression or storage of a composition description file in order to update the compositions groups in the global registry of services.

2) Fault notification to the global service registry: We can avoid the reuse of a failed service by updating the service state. The only component that can notify the service failure to the global registry is the message monitor. This corresponds to the action 3' in figure 7. The fault notification to the clients can be through the global service registry, or directly through the message monitor. The client entity will therefore proceed to update the service list displayed to users, for example by coloring in red the failed services. Avoiding the reuse of a

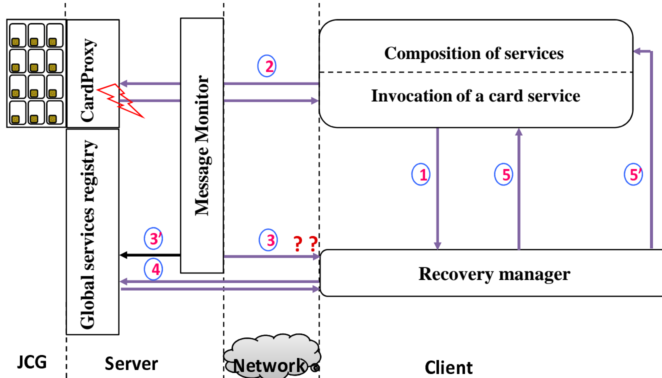

Fig. 7. Fault notification to the global service registry.

failed card service can also be seen as a handling of a previous fault in order to prevent it from happening again.

3) Access in mutual exclusion to the grid cards: In order to avoid the fault of the simultaneous access to different services in the same card, we allow only mutually excluded accesses to the different grid cards. This enables to ensure more secure and fast card service invocation. The access in mutual exclusion is applied by a mechanism of service reservation and liberation. If a card service is not ready to be reserved (failed, inexistent, or already active), the reservation module uses the interrogation module in order to find services that can substitute the unavailable one and it proposes them to the calling entity. This improves card service availability. Every card service reservation or liberation must be followed by an updating of the service state in the global registry, as well as by a notification to clients so that they update their service lists.

V. Performance Evaluation

We deployed a set of arithmetic services in the JCG and we evaluated the service execution time in the grid with improved dependability, compared to the service execution time in the original grid. We also evaluated the cost of tolerating a fault during the service execution. We didn't take into account neither the cost of the distant access to the grid nor the cost of securing the communication with the card services. We used a computer with a Pentium 4 processor 2.66 GHz, having 256Mo size of memory, and IBM JCOP20 [2] cards. We consider the service execution time the period separating the service invocation time and the result obtaining time.
We evaluated the execution time increase due to the improvement of the JCG dependability, varying the service invocation size (number of APDU commands sent to this service) and the number of clients connected to the grid. We invoked only one time, then 10 times and 25 times the method adding two integers of the card service *BasicCount*. For invocations containing more than an APDU command, we used the service composition tool of the platform. We noticed that the resulting overhead is very small (about 12%). This shows that the service reservation and liberation operations, added by the fault prevention layer, are very inexpensive. In fact, the invocation layer liberates the invoked service only after obtaining the invocation result. So, the time needed for the service liberation isn't included in its execution time. We also noticed that the overhead is inversely proportional to the invocation size. Since the service reservation is done only one time during the invocation process, more the invocation execution time increases more the influence of the time needed for the service reservation becomes less considerable. In addition, we remarked that increasing the number of clients connected to the grid didn't increase the execution time overhead. This result was expected since the notification to clients of a service reservation is made after sending the reservation result to the concerned client. Hence, the necessary time for the notification is not included in the service reservation time. Besides, on the server side, clients' requests to reserve services, to send APDU commands, or to search services for recovery have always priority with regard to the notification tasks.
In order to evaluate the fault tolerance cost, we invoked the method *short recurrent_series(short rank)* of the card service *Series* and we provoked a fault during this service execution. The method calculates the term having the rank entered as parameter, of the series $U_{n+1} = (U_n + U_{n-1})\ div\ 2$ with $U_0 = 32766$ and $U_1 = 1$. We measured the service execution time while varying factors that may influence the recovery cost such as the invocation execution time (by varying the

data size and the invocation size), the fault instant, and the recovery nature.
The results, when the recovery is by replication, showed that the overhead is not very small but in our opinion it is generally acceptable. In fact, we noticed that, for the invocation containing only one APDU command, the recovery cost is more important when the fault occurs at the end of the service execution. This is for the reason that after substitution, the service execution is not continued on the new card; it is rather triggered again on this card. For invocations containing more than one APDU command, the overhead values are smaller. A fault provoked at the end of the invocation can induce a relatively weak recovery cost because on the client side, the recovery concerns only the failed APDU command independently of the previous commands.
When the recovery is by composition, the failed service is substituted by the composition of the two card services *BasicCount* and *Division* implied by their respective methods *short sum (short,short)* and *short divide(short,short)*. The evaluation results showed a very big overhead especially when the two card services are held separately in two different cards. The fact that the cost of a recovery by composition is higher than the cost of a recovery by replication is expected since the failed APDU command is going to be replaced by several commands, and because of the time needed for the code generation from the composition description file. In the case where the new services are held in the same card, every APDU command sent to a service must be preceded by sending a Select command. Whereas, in the case where each service is installed in a Java card, selection commands are necessary only one time during the composition execution. This explains the fact that the recovery by composition of services held separately in different cards is more expensive than the recovery by composition of services held in the same card.

VI. Related work & background

The Java Card Grid Project (JCGP) [8], [11] focused on the security aspects of the JCG. In fact, using Java cards as the computing units on which applications and services are executed makes the grid a highly secured platform [3], [7], [10]. These Java cards provide secure data storage as well as secure code storage and execution. The JCGP also proposed a solution that improves the JCG storage capacity, still insuring a high security level [9]. In addition, communication security in the grid is realized by establishing a secure channel between the client application and the card service as well as between the cards themselves. To achieve this goal, JCGP members designed their own cryptographic protocol [15]. It is clear that all these works focused only on one attribute of the dependability which is the security attribute. Our work deals with other dependability attributes particularly service continuity and availability. In our knowledge, no prior work dealt with such dependability aspects in a JCG. Here, we propose an approach that preserves the security characteristics of the JCG and improves its dependability by fault tolerance and fault prevention techniques.

Dependability [4], [16] is made using four techniques: fault prevention, fault removal, fault forecasting, and fault tolerance. The three first techniques aim to assure fault avoidance. Since there exist unavoidable faults, fault tolerance technique [13] aims to deliver correct service despite the presence of these faults. Fault tolerance is generally implemented using error detection and subsequent system recovery which consists in fault handling and error handling. Fault handling prevents a located fault from being activated again and error handling prevents an error from leading to service failure. Error handling is usually achieved by using a replication of the system components or services. Two main replication policies are used: active replication [19], [20] and passive replication [6], [14]. In the active replication (also called the *state machine* replication), the client's request is received and processed by all replicas. Then, each replica sends a response to the client. This technique ensures a fast recovery. However, it uses processing resources heavily.
In the passive replication (also called the *primary-backup* replication), only the primary replica handles the client's request and provides a response, while the other secondary replicas are synchronized with the primary, using *checkpointing* (state transfer mechanisms). This technique uses less resources than active replication does. However, error handling is slower since it needs a new primary replica reelection and sometimes the request reprocessing by this new primary replica. For this reason, active replication is often considered a better choice for most real-time systems, and passive replication for most other cases [21].

VII. Conclusion

In this paper, we proposed a solution to improve Java Card Grid dependability, based on fault prevention and fault tolerance techniques. Fault tolerance is implemented by applying a passive replication strategy at card services level. Moreover, when a recovery by replication is impossible, our approach exploits possible compositions of card services with varied complexity degrees to substitute the failed service by a composition of other card services. Fault tolerance is transparent to the user and supports a recurrent recovery that allows to deal with eventual faults occurring during the handling of a previous error. Fault prevention procedures permit to prevent the use of an unavailable card service so that they minimize a recovery necessity. Thus, our solution ensures the grid service availability and continuity despite the presence of many faults that may occur in a JCG like platform.
The evaluation of execution times shows that the overhead of the added software layer is relatively acceptable either when there is no fault or when the recovery is by replication. However, the overhead is much bigger when the recovery is by composition.
For this fact, one of the perspectives of this work is to study the reasons of the recovery by composition cost increase and to find solutions improving this kind of recovery. We also aim to enrich the service card descriptor so that it contains indications about the quality of the service (execution time, reliability,). So, some more precise criteria may be defined to

better select card services for the recovery. Finally, we plan to generalize our approach to deal with faults related to stateful card services.

ACKNOWLEDGMENT

We would like to thank Achraf Karray for his valuable suggestions and advice.

REFERENCES

[1] *http://www.pcscworkgroup.com/.*

[2] PC/SC Workgroup, *http://www.zurich.ibm.com/JavaCard.*

[3] E. Atallah, S. Chaumette, F. Darrigade, A. Karray, and D. Sauveron. A grid of java cards to deal with security demanding application domains. In *6th edition e-Smart conference & demos, Sophia Antipolis, Frensh Riviera*, 2005. e-smart 2005 Isabelle Attali Award for the best innovative technology.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] M. B. Brahim, F. Baccar, A. Karray, M. B. Jemaa, and M. Jmaiel. Approche basee composition pour les applications sur une grille de cartes java. Septiemes Journees Scientifiques des Jeunes Chercheurs en Genie Electrique et Informatique, GEI'07, Tunisia 2007.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *In S.J. Mullender editor,Distributed systems (2nd Ed.), chapter 8, Addison Wesley*, pages 199–216, 1993.

[7] S. Chaumette, P. Grange, A. Karray, D. Sauveron, and P. Vigneras. Secure distributed computing on a java card grid. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 5*, page 186.2, Washington, DC, USA, 2005. IEEE Computer Society.

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

[9] S. Chaumette, A. Karray, and D. Sauveron. Extended secure memory for a java card in the context of the java card grid project. In *11th IEEE Nordic Workshop on Secure IT-systems: NordSec*, Oct 2006.

[10] S. Chaumette, A. Karray, and D. Sauveron. Secure collaborative and distributed services in the java card grid platform. In *CTS '06: Proceedings of the International Symposium on Collaborative Technologies and Systems*, pages 56–63, Washington, DC, USA, 2006. IEEE Computer Society.

[11] 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.

[12] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[14] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[15] A. Karray. Calcul securise sur grille de cartes a puce. Master's thesis, ENIS - University of Sfax, 2004.

[16] J.-C. Laprie. Dependability: Basic concepts and associated terminology. *Dependable Computing and Fault-Tolerant Systems series*, 5, 1992.

[17] S. microsystems. *Java CardTM 2.2.1 Specifications*. Sun microsystems, 2003.

[18] N. F. Noy and D. L. Mcguinness. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge System Laboratory, CA, 2001.

[19] F. Schneider. Replication management using the state machine approach. *In S.J. Mullender, editor, Distributed Systems, chapter 7. Addison Wesley*, pages 169–198, 1993.

[20] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[21] J. B. Sussman and K. Marzullo. Comparing primary-backup and state machines for crash failures. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 90, New York, NY, USA, 1996. ACM.