Extensible and Customizable Just-In-Time Security (JITS) Management of Client-Server Communication in Java*

Illustration on a simple electronic wallet

S. Chaumette[a] and P. Vigneras[a].
{Serge.Chaumette, Pierre.Vigneras}@labri.u-bordeaux.fr

[a]LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I, 351 Cours de la Libération,
33405 TALENCE, FRANCE

Security is becoming a more and more important issue when considering communication between clients and services. This is due to the critical aspects of the services and information that are now being dealt with (electronic wallet, private medical information, etc.) and to the fact that multi-applicative devices (Java cards [12], PDAs [4], etc.) are becoming of common use, what might give rise to some interest among the hackers. Over the air communication technologies such as Bluetooth[8,15], IRDA[24] or 802.11 [2] make the exchange of information even less secure, since the communication signal is straightforward to capture.

In this paper we present a framework that we have designed to provide what we call Just In Time Security (JITS) management of data exchange. This mechanism makes it possible to dynamically install a communication protocol between a client and a server, without the client or the server even noticing. It makes it possible to secure legacy services and to decide on the flight the security protocol to run.

Keywords : client server communication, security, EJBs, distribution, remote services, Java, e-commerce, legacy services.

1. INTRODUCTION

The work presented in this paper is carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI), Université Bordeaux 1 and CNRS. It takes place within the *Distributed Objects and Systems* (SOD) research team. The main characteristics of our activities is the focus on the Java[3] programming language, or more precisely on the features it provides. We use these features to set up and prove properties of operational distributed environments for use in the industry.

We are currently working on an *Extensible and Customizable Just-In-Time Security* (*JITS*) management system. The aim of the JITS is to enable the installation of a communication protocol in a dynamic manner between two systems that need to communicate, i.e. a client and a server. It might be the case that the protocol is unknown when the service is developed, and it may be chosen by the service, the client or even an external entity. This JITS is based and implemented on top of JACOb[21,22]. JACOb is a system close in some sense to the Enterprise Java Beans[5,19] and Jini[17,18]. It has been developed in our team. It makes it possible to run communicating distributed components and to make legacy services available through the network, i.e. remotely.

Throughout this paper, we will illustrate our mechanism on an electronic wallet, that we made as simple as possible for the sake of explanation. The methods of this service are shown in the server side of figure 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.

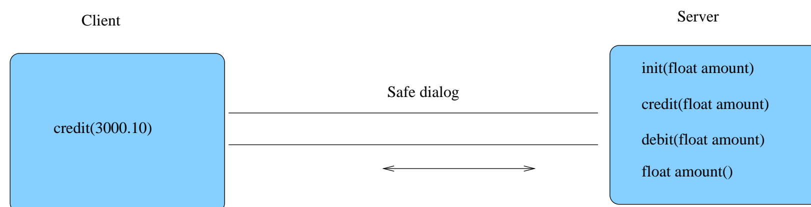

Figure 1. A basic electronic wallet service and its client

2. RATIONAL FOR A JITS

The framework we offer supports features that we believe are enabling technologies to provide wide spread exchange of sensible information over the various networks, either wired or over the air. In this section we first describe the main problems one has to face regarding the security of communication. Based on it, we then build the rational for the JITS framework that we have set up.

2.1. Analysis of the problems

– **Problems related to standards.**

There are some characteristics of communication security standards that must be taken into account so as to propose an open security framework:

- the wide number of available algorithms and techniques prevents from definitely choosing a specific one, and therefore requires adaptability;
- it is sometimes difficult to evaluate the available algorithms properly. It may be the case that algorithms now considered robust become obsolete tomorrow : such algorithms have to be replaced by more robust ones;
- new algorithms are invented regularly and their must be room to integrate them in pre-existing software environments.

– **Problems related to technology.**

It is more and more often the case that protocols are wired in the hardware; new cards will require new drivers and protocols to be installed in a dynamic manner.

– **Problems related to users.**

For psychological rather than technical reasons, users will not necessarily trust security layers provided by someone else. They will prefer to install their own usual communication system to communicate with a given service.

2.2. Necessary features for a JITS

From the above analysis, we can exhibit a set of constraints that have to be met by a JITS. Here are the most important:

- support for future protocols that are not even identified;
- support for the selection of the protocol by the service, the client or even an external entity;
- support for dynamic management of the set of communication protocols available for a given service.

3. THE JITS FRAMEWORK

3.1. Related work and the base architecture

We are currently building our framework on top of an environment that we have designed and implemented. This is called JACOB[22,21] for *Java Active Container of Objects*. The current release of JACOB makes it possible to make remote any object, any service, without any need for compilation, dedicated key words or declarations in the object being dealt with. One of the direct consequences of that point is that it supports legacy code. We believe that the environment that we propose does not really compare to other existing projects, although the system it relies upon, JACOB [22,21], can be compared by certain aspects to Jini[17,18] or Enterprise Java Beans[5,19], all of these systems offering a framework and a software run-time environment that makes it possible to run communicating distributed components. We insist on the fact that the main fundamental difference when comparing JACOB to other systems is that there is no need to modify the code of an object to make it accessible remotely.

Furthermore, within usual systems, once a service has been developed there is no way to change the protocol that it uses to communicate with its clients: this feature is one of the main characteristics offered by our JITS framework. Objects or services do not need to be developed specifically to embed network access features as it is the case when using Java RMI[16], CORBA[14,23], DCOM[11] or similar software environments. The service is integrated inside an active container (figure 2) that offers a minimal interface (a `call` method that uses reflection [1]) to invoke the methods of the objects it contains. So that objects can be accessed remotely, it is sufficient that the active container is accessible remotely, what is possible because JACOB supports this feature – in deed it supports separation of concerns and aspect[9,10] programming.

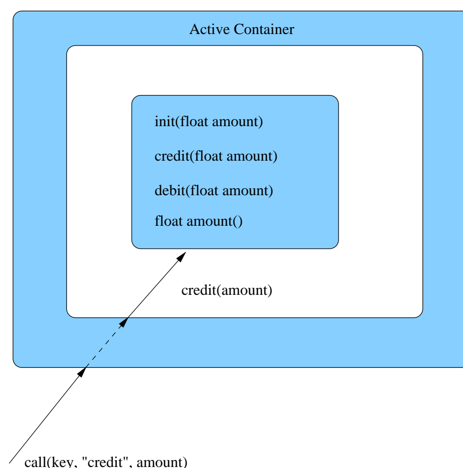

Figure 2. A service inside an active container

Of course, a client side proxy can be developed (figure 3) to support the same interface on both server and client side. This implies that this proxy necessarily uses the communication protocol imposed by the active container, since it is its peer for the communication - the active container then forwards calls to the effective service -. We prefer the solution where both a client side and a server side proxy are used as described in the next section.

3.2. Building the JITS on top of JACOb

The technique that we use to implement our JITS consists in setting up what we call a *protocol manager*. This manager is in charge of managing the communication environment for a given

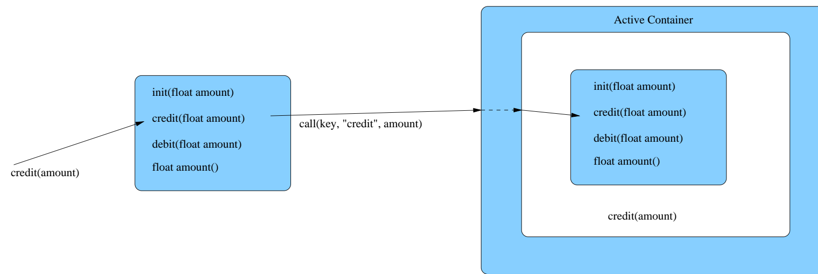

Figure 3. A client side proxy

service (figure 4). This protocol manager creates an intermediate object, that we call a *server side proxy*, that is in charge of the communication with the client, and that is in charge of implementing the selected security protocol.

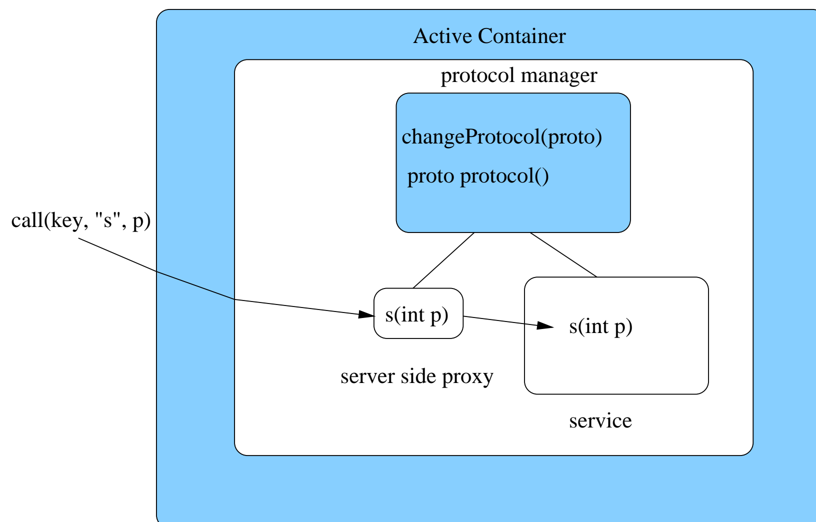

Figure 4. A service, its protocol manager, and a server side proxy

We now describe the steps required to effectively use the JITS framework (see figure 5).

Step 1.

An active container (1) must be running. A service can also be running (2), possibly inside the active container.

Step 2.

The client application uploads a protocol manager (3) to handle the given service inside the active container.

Step 3.

The client requests the creation of an instance of a given protocol proxy (4), possibly uploading a protocol handler to the protocol manager.

Step 4.

It gets as a result a client side proxy (5) to communicate with the service, or more precisely with the server side proxy of the service.

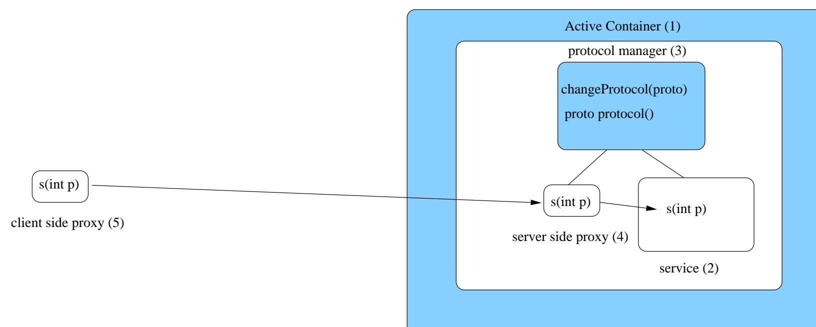

Figure 5. Client side proxy, server side proxy and protocol manager

The service and the client have no idea about the protocol that is being used to communicate between the two proxies. Many possibilities are available depending on the goal of the application or of the service or on some external policy: direct communication between the client side proxy and the server side proxy; communication through the `call` method of the active container in order to achieve some additional operation possibly implemented by the active container (transaction, authentication, etc.). This last solution furthermore makes it possible to implement generic proxies and then generic protocol managers.

The client and the server can now communicate using the protocol that has been dynamically installed.

4. UNDERLYING FORMAL MODEL

The concept of active container relies upon a strong foundation. We have a π-calculus [13] formal model [20] of their implementation and operation. We believe that this model will help in the process of proving and hence convincing potential users of the reliability of the features provided by our system.

This model is based on π-calculus because it provides support for modeling the migration of code and of communication channels. It is close to the effective implementation, what is a plus to us.

The formal model of our system is out of the scope of this paper and it will not be detailed here.

5. CONCLUSION AND FUTURE WORK

We believe that the system we propose solves the problem of making legacy services more secure. It also provides an appropriate environment to support security for new services. The current implementation is based on Java but nothing prevents the services from being developed in any language.

Of course, we also need to secure the basic bricks of the system itself, i.e. JACOb. We will use the security features provided by the Java language [7] and by JAAS [6], the Java Authentication and Authorization Service. We are in the process of porting an implementation of our JITS on iPAQ

PDAs with Bluetooth communication so as to demonstrate its use for over the air communication between PDAs.

REFERENCES

[1] Java core reflection: Api and specification, 1997.
[2] The working group for wireless local area networks, 2001.
[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, 2000. Third edition.
[4] C. Corporation. Com palmpilot.
[5] R. Englander. *Java Beans, Guide du programmeur*. O'REILLY, 1997.
[6] C. L. L. Gong, L. Koved, A. Nadalin, and R. Schemers. User Authentification and Authorization in the Java Platform. Technical report, Sun Microsystems, 1999.
[7] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999. ISBN: 0-20131-000-7.
[8] J. Haartsen. The bluetooth radio system, 2000.
[9] Timothy Highley, Michael Lack, and Perry Myers. Aspect oriented programming: A critical analysis of a new programming paradigm. Technical Report CS-99-29, 5, 1999.
[10] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
[11] Microsoft Corporation. DCOM Technical Overview. Technical report, Microsoft Corporation, 1996.
[12] S. Microsystems. Java card technology, 2000.
[13] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, Cambridge, UK, 1999.
[14] J. Siegel. *CORBA, Fundamental and Programming*. Wiley, 1996.
[15] T. Special and I. Group. Specification of the bluetooth system.
[16] SUN Microsystems. *Java Remote Method Invocation Specification*, 1998.
[17] SUN Microsystems. Jini Architectural Overview. Technical report, Sun Microsystems, January 1999.
[18] SUN Microsystems. A Collection of Jini Technology Helper Utilities and Services Specifications. Technical report, Sun Microsystems, October 2000.
[19] SUN Microsystems. *Enterprise JavaBeans Specification*, juin 2000. Version 2.0, Public Draft.
[20] P. Vignéras. Agents mobiles : une implémentation en Java et sa modélisation, Juin 1998. LaBRI, Université Bordeaux I, DEA Report.
[21] P. Vignéras. Jacob : un support d'exécution pour la distribution d'objets en Java. In *JCS'2000*. RenPar'2000, June 2000.
[22] Pierre Vignéras. Jacob : a software framework to support the development of e-services, and its comparison to Enterprise JavaBeans. In *Perspectives for Commercial and Scientific Environments*, pages 11–12, University of Technology Munich, 19-20 April 2001. International Workshop on Performance-Oriented Application Development for Distributed Architectures (PADDA).
[23] S. Vinoski. CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Comunications Magazine*, 35(2), February 1997.
[24] S. Williams. Irda: Past, present and future, 2000.