Middleware Support for Resource-Constrained Software Deployment

Nicolas Le Sommer, Frédéric Guidec

Middleware Support for Resource-Constrained Software Deployment

Nicolas Le Sommer and Frédéric Guidec

VALORIA Laboratory
University of South Brittany, France
{Nicolas.Le-Sommer, Frederic.Guidec}@univ-ubs.fr

Abstract The JAMUS platform is dedicated to providing a safe runtime environment for untrusted Java application programs, while offering each of these programs access to the resources it needs to use at runtime. To achieve this goal, JAMUS implements a contractual approach of resource utilisation and control, together with a reservation-based resource management scheme, and a monitoring model. When getting deployed on the platform, a candidate program must first subscribe a contract with the resource manager of the platform. This contract describes the resources the program requires to be able to access at runtime, and how it plans to behave when accessing these resources. Based on this information, the platform can monitor programs at runtime, so that any violation of the contracts they subscribed can be detected and sanctioned. Moreover, since the specific needs of a program are liable to change (or to be refined) dynamically while this program is running, any program hosted by the platform can ask that its contract be re-negotiated at any time.

1 Introduction

The growing popularity of mobile and network-dependent Java application programs leads to an increased demand for deployment platforms that permit to launch and run potentially dangerous Java application programs (such as programs downloaded from untrusted remote Internet sites) in a restrained environment.

Sun Microsystem's platform *Java Web Start* was designed in order to meet this demand. Yet, because this platform relies on the security model of the standard *Java Runtime Environment* (JRE) platform, it shows a number of limitations. For example, security in *Java Web Start* can only be obtained by restraining the access to a strictly pre-defined set of system resource types (namely, network sockets and files). Moreover, security is based solely on access permissions. In our opinion, this approach does not permit sufficient control over the behaviour of application programs.

With JAMUS *(Java Accommodation of Mobile Untrusted Software)* we give some solution to the above-mentioned. Resource access control in JAMUS can be applied to a larger (and easily extensible) variety of resource types (including the CPU and memory), and this control can be performed at a finer grain (for example, restrictions can be imposed of the amount of CPU time or memory consumed by each Java thread).

JAMUS implements a contractual approach of resource management and access control. Application programs are expected to evaluate and specify their own needs regarding the resources they wish to use at runtime. Based on this information, the platform

can decide whether a candidate program should be accepted or rejected. Moreover, when a program has been accepted for running on the platform, its behaviour is monitored, so that any violation of the contract it subscribed with the platform can be readily detected and dealt with.

It is worth mentioning that JAMUS is not exclusively dedicated to enforcing a security policy. It also strives to meet the specific requirements of each of these programs. To achieve this goal the platform implements a resource-reservation scheme. Whenever a program subscribes a contract with the platform, resources are actually reserved for this program from the platform's viewpoint.

The remainder of this paper is organised as follows. Related work is presented in Section 2, which also discusses some of the limitations we observed in other security-oriented projects and middleware platforms. Section 3 presents the general architecture of the JAMUS platform. Section 4 focuses on resource contracting. It shows how contracts can be defined by application programs, and subscribed with the platform. It also shows how the information contained in contracts is used by a resource broker, which implements a reservation scheme in order to satisfy the programs' requirements. Section 5 shows how JAMUS was implemented on top of RAJE *(Resource-Aware Java Environment)*, an object-oriented framework we designed, and which provides many facilities for resource monitoring and control in Java. Section 7 concludes this paper, enumerating some of the topics we plan to address in the near future.

2 Lessons learned from related work

Security in the Java Runtime Environment (JRE) relies on the so-called "sandbox" model. With this model, any program runs in a restrained environment, whose boundaries somehow define the evolution space of this program. In the first versions of the standard JRE (as proposed by *Sun Microsystems*), two alternative configurations were defined. On the one hand, plain Java applications were considered as safe code, and were therefore granted full access to system resources (such as network sockets and files). On the other hand, Java applets (ie code downloaded from remote Internet sites) were considered as potentially malicious pieces of code, and were only granted access to a strictly limited subset of resources [6]. With later releases of the Java platform (up to the current Java 2 platform), this simplistic security model was extended in order to implement the concept of protection domain [7,6]. A protection domain is a runtime environment whose security policy can be specified as a set of permissions.

The security model implemented in the traditional JRE relies on stateless mechanisms. A major limitation of this approach is that access to a specific resource cannot be conditioned by whether the same resource was accessed previously, or by how much of this resource has already been consumed. As a consequence, quantitative restrictions (such as shares of CPU time, or I/O quotas) cannot be set on the resources accessed from protection domains. With this limitation, the security mechanisms implemented in the JRE cannot prevent an over-consumption of resources, such as that observed in denial of service attacks, or with many faulty program codes.

In our opinion, a safe deployment environment for programs of dubious origin should allow that access restrictions be specified in both qualitative and quantitative terms (ie access permissions and access quotas). Environments such as JRes [4], GVM [2] and KaffeOS [1] extend the traditional JRE along this line. They implement mechanisms that make it possible to evaluate how much memory and CPU time is used by an active entity (a thread for JRes, a process for GVM and KaffeOS). With JRes, one can additionally count the number of bytes sent and received through the network.

Although JRes, GVM, and KaffeOS provide advanced facilities for resource consumption accounting, this accounting is only possible at a rather coarse grain. For example, network access accounting in JRes is performed on a per thread basis. Consequently, JRes cannot count the number of bytes exchanged with a given remote host, or with a specific remote port number. Yet, we consider that such fine-grain accounting would be an advantage when deploying untrusted programs, as it would permit the definition and the enforcement of very precise security policies.

Naccio [5] and Ariel [9] are projects that permit such fine-grain resource access accounting. They both define a language for defining a security policy, together with mechanisms for enforcing this policy while an application program is running. Security policy enforcement is carried out statically, by rewriting the application program bytecode, as well as that of the standard Java API. An advantage of this approach is that the cost of the supervision of a program is kept at a minimum since code segments that check the access to a particular kind of resource are inserted in Java API classes only if the selected security policy requires it. However, the generation of an API dedicated to a specific security policy is a quite expensive procedure. The approach proposed in Naccio and Ariel is thus mostly dedicated to the generation of predefined Java APIs that each enforce a generic security policy.

In contrast, our work aims at providing each application program with a runtime environment that fits its needs perfectly. In other words, we wish to launch each program in a sandbox whose boundaries are defined based on information provided by the program itself. Moreover, it must be possible to modify the boundaries of this sandbox dynamically, thus altering the restrictions imposed on the program's behaviour. This condition is motivated by the observation that, in some circumstances, the exact needs of a program can hardly be defined statically (consider for example a program that is meant to read a file whose name will only be discovered at runtime). Another reason is that the needs of a program are often liable to change at runtime, especially if this program must run for a long time.

3 Overview of the JAMUS platform

The architecture of the JAMUS platform is shown in Figure 1. This figure shows many elements, which cannot all be described in details in this paper for the sake of brevity. Indeed, this paper most specifically focuses on those elements of the figure that directly pertain to contract negotiation and contract monitoring. These topics are discussed further in the remainder of this section, and details about their implementation are given in the next sections.

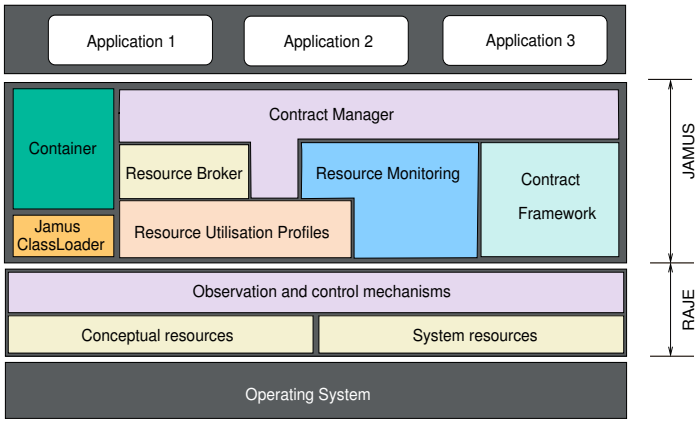

Figure 1. Architecture of the JAMUS platform.

3.1 A contract-based approach of resource control

JAMUS implements a contractual approach of resource utilisation and control. Before getting launched on the platform, a candidate program must first subscribe a contract with the contract manager of the platform, thus informing this manager about the resources it plans to use at runtime. The contract manager is responsible for binding contracts with candidate programs, and for the registration of all running contracts. It itself relies on a resource broker, whose role is to keep track of the resources available on the platform at any time, to perform some admission control on behalf of the contract manager (*eg* examining submitted contracts and deciding if they can be accepted), and to reserve resources for admitted programs. The implementation of the contract manager and that of the resource broker are detailed in Section 4, which also gives examples of typical interactions between these elements at runtime.

Since the exact needs of a program are liable to change at runtime, any program running on the platform can ask that its contract be re-negotiated as and when needed. Of course, in such a case the contract manager and resource broker are involved again, as they must decide if the modified contract can still be supported by the platform.

3.2 Security through contract monitoring

The contract a program subscribes with the platform effectively defines the boundaries of the sandbox in which this program should be launched and allowed to run. It also implicitly defines how this program should be expected to behave with respect to the resources offered by the platform. The information contained in a program's contract can thus be used at runtime to monitor this program, and to detect any violation of its contract.

Notice that contract monitoring would not be necessary if the programs deployed on the JAMUS platform could all be considered as trustworthy. If that was the case,

then any program could reasonably be expected to behave exactly as promised, that is, to access only those resources it required explicitly when subscribing a contract with the platform. However, JAMUS is dedicated to accommodating application programs of dubious origin, such as programs downloaded from untrustable remote Internet sites. Consequently, any application program deployed on the platform must be considered as a potential threat throughout its execution. Once a program has been accepted by the contract manager of the platform, the behaviour of this program must be monitored, so that any violation of its contract can be detected and sanctioned.

Contract monitoring in JAMUS relies on facilities we implemented and assembled in RAJE *(Resource-Aware Java Environment)*, an open and extensible Java-based framework for resource consumption monitoring and control. RAJE is discussed further in Section 5.

4 Resource contracting

4.1 Specification of resource access conditions

JAMUS implements an object model that makes it possible to reify a program's basic requirements or a platform's restrictions as so-called "resource utilisation profiles". In this model, an instance of class *ResourceUtilisationProfile* is meant to define specific access conditions to a particular resource –or collection of resources– in both qualitative and quantitative terms (eg access permissions and quotas). Basically, a *ResourceUtilisationProfile* object simply aggregates three objects that implement the *ResourcePattern*, *ResourcePermission*, and *ResourceQuota* interfaces respectively. JAMUS provides specific implementations of these interfaces for each basic resource type (Socket, DatagramSocket, File, Thread, CPU, etc.).

```
int MB = 1024*1024;
ResourceUtilisationProfile P1, P2, P3,;
P1 = new ResourceUtilisationProfile(
            new SocketPattern("http://www.music.com"),
            new SocketPermission(SocketPermission.ALL),
            new SocketQuota(15*MB, 1*MB));
P2 = new ResourceUtilisationProfile(
            new FilePattern("/opt/music"),
            new FilePermission(FilePermission.WRITE_ONLY),
            new FileQuota(0, 40*MB));
P3 = new ResourceUtilisationProfile
            new MemoryPattern(),
            new MemoryPermission(MemoryPermission.ALL),
            new MemoryQuota(2*MB));
```

Figure 2. Definition of resource utilisation profiles.

Figure 2 shows how resource utilisation profiles can be defined in Java. By including a given type of *ResourcePattern* in a *ResourceUtilisationProfile,* one indicates that

the access conditions defined in this profile are only relevant for those resources whose characteristics match the pattern. For example, the *SocketPattern* in profile *P1* (see Figure 2) indicates that this profile defines conditions for accessing the specified Internet site through a socket-based connection. The *SocketPermission* and *SocketQuota* objects combined in profile *P1* bring additional information: they indicate that once a connection has been established with the remote site, the amounts of bytes sent and received through this connection should remain below the limits specified. The other two profiles similarly specify conditions for accessing a given directory in the filesystem (*P2*), and for consuming memory (*P3*).

By defining profiles such as those shown in Figure 2, an application program can specify its own requirements regarding the resources it plans to use at runtime. For example, by inserting profiles *P1*, *P2* and *P3* in a contract, a program may simultaneously require access to a remote Internet site (with specific access permissions and quotas) and to a given part of the filesystem (again with specific access permissions and quotas), while requiring a certain amount of memory for its sole usage.

Besides serving as a way to describe application programs' requirements, resource utilisation profiles are also used in JAMUS to set limitations on the resources the platform must offer. At startup the resource broker of the platform is given a collection of resource utilisation profiles, that describe which resources should be made available to hosted programs, and in what conditions.

Since JAMUS must be able to negotiate contracts dynamically with Java application programs, we decided that contracts and resource utilisation profiles (which actually serve as contract clauses) should be considered as first-class objects in our model (as suggested in [3]). This is the reason why JAMUS relies on an object model in which contracts and profiles can be defined and managed directly as Java objects. Yet, we also defined an XML dialect for specifying profiles in a more human-readable form, and for storing this kind of information in files. JAMUS implements routines for parsing an XML file, and for instantiating profiles contracts based on the information found in this file. The administrator of a JAMUS platform can thus configure and manage this platform, without ever writing any source code.

4.2 Contract definition

Any resource utilisation profile defined by a program can be considered as expressing either a *real requirement* of the program (meaning, the program actually requests guarantees about the accessibility of the resource considered in the profile), or as a simple *indication* that the program *may* attempt to access the resource considered at runtime (meaning, the program simply provides information about the boundaries of the sandbox in which it wishes to be launched).

Since the resource broker of the platform implements a reservation scheme, we decided to distinguish between a program's true requirements (those that call for both resource reservation and access monitoring), and simple indications of the program's planned behaviour (that call for monitoring only).

Contracts in JAMUS are thus defined as collections of resource utilisation profiles, but these profiles are assembled in two distinct sets. Profiles in the first set call explicitely for resource reservation, whereas profiles in the second set do not.

Figure 3 shows the definition of two alternative contracts, that each combine several profiles. The first contract in this figure combines profiles *P1, P2* and *P3*, but only the first two profiles should be considered as real requirements (meaning, they call for resource reservation, whereas *P3* does not). The second contract combines profiles *P1*, *P4*, and *P5*, but only *P1* calls for resource reservation.

A candidate program can thus instantiate one or several alternative *Contract* objects, and submit these contracts to the platform. If one of the contracts is declared admissible by the resource broker, then this contract can be subscribed by the program. If several of the contracts are declared admissible by the resource broker, then the application program can additionally choose which of these contracts it wishes to subscribe.

```
Contract contract1 = new Contract ({P1, P2}, {P3});
Contract contract2 = new Contract ({P1}, {P2, P3});
```

Figure 3. Definition of two possible contracts that each combine several resource utilisation profiles.

4.3 Contract negotiation

Contract negotiation in JAMUS is performed as a two-step process. In the first step, several alternative contracts can be submitted to the platform by the same program. In the second step, one of the contracts the platform has approved (assuming there is at least one) must be selected by the program, and subscribed with the platform. Each contract is examined by the resource broker, whose role it is to decide whether a contract is acceptable or not, and to reserve resources when needed. Resource reservation is only achieved (if and where needed) by the resource broker when a contract is subscribed by a program. By reserving resources (or shares of a resource) a program obtains guarantees that its requirements will be satisfied at runtime.

Figure 4 shows a possible sequence of interactions between a program and the platform's contract manager. It also shows that the resource broker is consulted whenever the contract manager receives a contract submitted by the program. In this example the first contract submitted by the program is rejected by the platform. Such a negative reply might be justified by the detection of a conflict between one of the program's requirement and one (or several) of the platform's restrictions. Notice that whenever a contract is rejected by the resource broker, the candidate program is returned a detailed report that specifies which profiles in the contract could not be accepted by the platform. This kind of information is expected to be useful for candidate programs that are capable of choosing between several behavioural scenarios, or for programs that can adjust their demand about resources based on information returned by the platform.

In Figure 4 the second contract submitted to the platform is accepted by the resource broker. The candidate program can try to subscribe this contract. However, since the platform may be carrying out several negotiations concurrently with as many candidate programs, the status of available resources may change between the time a submitted

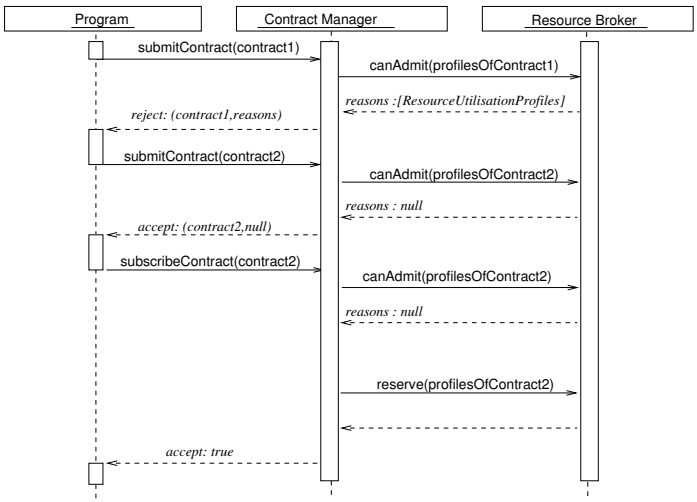

Figure 4. Sequence of interactions between a candidate program and the platform's contract manager and resource broker.

contract is declared acceptable by the resource broker, and the time this contract is subscribed. Consequently, whenever a program subscribes a contract, the terms of this contract are examined again by the resource broker in order to check whether they are still valid. If so, then the resources required by the candidate program are reserved for this program, as explained in the next section.

The reason why contract submission and contract subscription have been differentiated in the JAMUS platform is that it makes it possible for a candidate program to request that the platform examine several possible contracts (corresponding to different alternative combinations of resource requirements), before the program eventually decides which of these contracts it actually wishes to subscribe with the platform. This approach is meant to foster the development of application programs that can adjust their behaviour (at launch time or dynamically), depending on the resources the platform can offer.

5 Contract Monitoring

5.1 Security through resource monitoring

As mentioned in Section 3, any program hosted by the JAMUS platform is considered as not being trustworthy. As a consequence, the platform must monitor all running contracts in order to detect and to sanction any violation of a contract. Since contracts all pertain to the utilisation of resources, monitoring contracts actually comes down to monitoring the way resources are accessed and used by hosted programs.

JAMUS is dedicated to hosting Java programs, which run in a virtual machine. In this context, resource monitoring implies that all resources are reified as objects in the

JVM. To achieve this goal, JAMUS relies on the facilities offered by RAJE, an open and extensible framework we designed in order to support the reification and the control of any kind of resource using objects in Java (see Figure 1).

RAJE can be perceived as an extension of the traditional runtime environment of the Java 2 platform. It relies on a modified version of the standard JVM Kaffe (version 1.0.7), which allows the accounting of memory consumption and CPU time consumed by each Java thread. Some classes of the standard Java API (such as *Socket*, *DatagramSocket*, *File*, and *Thread*) were augmented so that any access to the resources they model can be monitored at runtime. New classes were defined in order to model system resources, such as the CPU, system memory, and system swap. Part of the code implemented in RAJE thus consists of native code that permits the extraction of information from the underlying OS, and the interaction with inner parts of the JVM.

More details about the facilities implemented in RAJE (including implementation details) can be found in [8].

By implementing JAMUS on top of RAJE, resource monitoring and control can be performed in JAMUS at a very fine grain. For example any socket or file opened by a Java program can be considered as a specific resource, and any access to such a resource can be accounted for, and restrained if needed. Any Java thread can likewise be considered as a specific resource, and the amounts of CPU time or memory space consumed by each Java thread can be monitored as well. RAJE also makes it possible to set locks on resources, thus preventing any further utilisation of these resources by application programs.

5.2 Component monitors and resource monitors

Every application program hosted by the JAMUS platform runs under the control of a dedicated component monitor. This monitor uses the resource utilisation profiles contained in the contract subscribed by the program in order to instantiate many resource monitors. Their mission is to monitor the utilisation of the resource –or collection of resources– considered in this profile, and to ensure that this utilisation conforms to the access permissions and quotas defined in the profile.

JAMUS provides a specific implementation of a resource monitor for each basic resource type considered to date in RAJE. Each resource monitor admits a resource utilisation profile as a creation parameter. The role of a resource monitor is to supervise the utilisation of the resource –or collection of resources– considered in this profile, and to ensure that this utilisation conforms to the access permissions and quotas defined in the profile.

The figure 5 shows how the resource monitors, the component monitors, the contract manager and the resource broker interact.

When a resource monitor detects a contract violation, it reports to the component monitor, which in turn applies the sanction defined in the platform's configuration. In the current implementation of the platform, several kinds of sanctions are actually applicable to faulty programs. These sanctions range from a simple warning addressed to a faulty program (using an event based model), up to the immediate termination of this program.

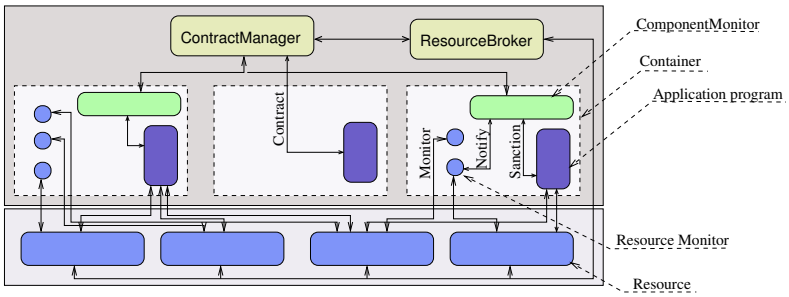

Figure 5. Monitoring of applications

6 Performance results

When designing a deployment platform such as JAMUS, one can legitimately worry about the overhead imposed by dynamic sandboxing and contract monitoring.

In order to evaluate how these mechanisms can impact on the performances of the application programs launched on the platform, we recently initiated an evaluation process. This process consists in running a series of demanding programs (that is, programs that use resources extensively), while measuring their performances in different conditions.

For example we launched an FTP server (written in pure Java code) in JAMUS, and we measured the network throughput observed while downloading large files from this server. This experiment was conducted using two workstations (2.4 GHz Pentium 4 processor, 512 MB RAM) connected by a Fast Ethernet link (100 Mbps, Full Duplex). The throughput observed during file transfers was measured when running the FTP server with two standard JVMs (IBM's and Kaffe), and with JAMUS (which relies on a modified version of Kaffe). Moreover, in the latter case the FTP server was launched with a varying number of requirements, so that at runtime its behaviour was observed by a varying number of resource monitors (typically one monitor for filesystem access, and one or several monitors for network access).

JVM	Throughput (Mbps)
Kaffe (version 1.0.7)	89.5 *(100 %)*
IBM JVM (version 1.4.1)	89.3 *(99.8 %)*
JAMUS (no monitor)	88.9 *(99.3 %)*
JAMUS (2 monitors)	86.3 *(96.4 %)*
JAMUS (3 monitors)	84.6 *(94.5 %)*
JAMUS (5 monitors)	81.9 *(91.5 %)*

Table 1. Performances observed with an FTP server running either in a standard JVM or in JAMUS (with a varying number of monitors).

The throughputs we observed are reported in Table 1. In this table the throughput observed with the standard JVM Kaffe is used as a reference value.

We consider that these results are quite satisfactory. Obviously the monitoring infrastructure implemented in JAMUS significantly alters the performances of the application programs launched in this platform. Yet, in our opinion the degradation of performances observed while running the FTP server (which is a quite demanding program as far as filesystem and network resources are concerned) remain acceptable.

Besides, it is worth mentioning that the source code pertaining to resource consumption accounting in RAJE, and to contract monitoring in JAMUS, was primarily developed so as to be readable and flexible. Parts of this code could probably be written differently, though, in order to reduce the overhead imposed on the programs launched on the JAMUS platform.

7 Conclusion

In this paper we have presented the architecture and the implementation of the JAMUS platform, which is dedicated to hosting untrusted Java application programs, provided that these programs can specify their own needs regarding the resources they plan to use at runtime.

JAMUS actually constitutes a demonstrator platform, with which we experiment with the idea of "resource-aware" programs and deployment platforms, that is, programs that can identify and specify their own needs regarding resource utilisation, and platforms that can use this kind of information in order to provide such programs with differentiated services. Although JAMUS is specifically dedicated to hosting non-trustable application programs, it is our conviction that many other application domains and systems (such as agent-based systems, or adaptive systems) could benefit of — or take inspiration from— the models and mechanisms we develop in this particular context.

References

1. Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *The 4th Symposium on Operating Systems Design and Implementation*, October 2000.
2. Godmar Back, Patrick Tullmann, Legh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the Design of Java Operating Systems. In *USENIX Annual Technical Conference*, June 2000.
3. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract-aware. In IEEE, editor, *Computer*, page 38 44. IEEE, June 1999.
4. Grzegorz Czajkowski and Thorsten von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.
5. David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy*, May 1999.
6. Li Gong. Java Security: Present and Near Future. *IEEE Micro*, -:14–19, May 1997.
7. Li Gong and Roland Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Scurity*, March 1998.

8. Frédéric Guidec and Nicolas Le Sommer. Towards Resource Consumption Accounting and Control in Java: a Practical Experience. In *ECOOP'2002 (Workshop on Resource Management for Safe Languages)*, June 2002. To be published.
9. Raju Pandey and Brant Hashii. Providing Fine-Grained Access Control for Java Programs. In *The 13th Conference on Object-Oriented Programming, ECOOP'99*. Springer-Verlag, June 1999.