A Contract-Based Approach of Resource-Constrained Software Deployment

Nicolas Le Sommer, Frédéric Guidec

A Contract-Based Approach of Resource-Constrained Software Deployment

Nicolas Le Sommer and Frédéric Guidec

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

Abstract Software deployment can turn into a baffling problem when the components being deployed exhibit non-functional requirements. If the platform on which such components are deployed cannot satisfy their non-functional requirements, then they may in turn fail to perform satisfactorily. In this paper we propose a contract-based approach of resource-constrained software deployment. We thus focus on a specific category of non-functional requirements: those that pertain to the resources software components need to use at runtime. Ultimately, our objective is to provide software components with means to specify their requirements regarding hardware and/or software resources, and to design methods and models for utilising this kind of information at any stage of a component's life-cycle. The paper reports the design of JAMUS, an experimental platform we develop in order to support the deployment of mobile software components, while providing these components with guaranteed access to the resources they need. JAMUS implements a contract-based model so as to recognise and to allow for the requirements of components regarding resource access and consumption.

1 Introduction

When deploying software components on a target platform, one can wonder whether the requirements of these components can be satisfied by the runtime environment offered by the platform. Almost any software component exhibits specific non-functional requirements. Such requirements can in turn be categorised, depending on whether they refer to properties such as persistence, fault-tolerance, reliability, performance, security, etc. As observed in [17], 'it is obvious that in most practical examples a violation of non-functional requirements can break [components] just as easily as a violation of functional requirements'. Hence, if the platform on which a component is deployed fails to meet its non-functional requirements, then the component may in turn fail to fulfil its mission, which usually consists in providing a clearly defined set of services with a certain level of quality of service (QoS). The question of deciding whether the non-functional requirements of components can be satisfied is even more crucial on an open platform, for new components can be deployed on – or removed from – such a platform while other components are running. Dynamically loading or removing a single software component on such a platform may impact dramatically on all the components that share the same runtime environment.

In this paper we focus on a specific category of non-functional requirements, that is, those that pertain to the resources a software component needs to use at runtime. All software components are not equivalent as far as resource access and consumption are concerned. Some components can do very well with sparse or even missing resources, while others require guaranteed access to the resources they need. In order to deal with non-functional requirements pertaining to resource utilisation we propose a contractual approach of resource management and access control. This idea is being investigated in the context of project RASC[1] *(Resource-Aware Software Components)*. This project aims at providing software components with means to specify their requirements regarding hardware and/or software resources, and to design methods and models for utilising this kind of information at any stage of a component's life-cycle.

This paper mostly reports the design of JAMUS, an experimental platform whose development is in progress within project RASC. The JAMUS platform supports the deployment of software components, provided that these components can specify their requirements regarding resource utilisation in both qualitative (eg access rights to parts of the file system) and quantitative terms (eg read and write quotas). JAMUS is actually most specifically dedicated to hosting simple mobile Java components, such as application programs and applets. These could be downloaded from remote Internet sites or received as Email attachments before being deployed on the platform. As a consequence, emphasis is put in JAMUS on providing a safe and guaranteed runtime environment for components of such dubious origin. ("JAMUS" is actually an acronym for *Java Accommodation of Mobile Untrusted Software*. Any mobile component deployed on this platform is considered as a potential threat throughout its execution.)

The remaining of this paper is organised as follows. Section 2 presents the general architecture of the JAMUS platform. It also gives an overview of RAJE, the runtime environment this platform relies on, and whose development is also carried out within project RASC. Section 3 introduces the notion of resource utilisation profile, and it shows how this notion is implemented and used in JAMUS. Resource utilisation profiles are important elements in the platform, as they make it possible to set restrictions on the resources it offers, while specifying the requirements of hosted components regarding these resources. The notion of resource usage profile is thus the key to resource contracting in JAMUS. This specific mission is under the responsibility of a resource broker, whose role and main features are detailed in Section 4. Since the mobile components deployed on the platform are perceived as inherently non-trustworthy, their execution is submitted to a constant monitoring, so that resource access violations can be readily detected and dealt with. Section 5 details the mechanisms that permit this monitoring in JAMUS. Related work is mentioned in Section 6. As a conclusion, Section 7 discusses some of the current limitations of the JAMUS platform, and gives an overview of the lines we plan to work along in the future.

[1] `http://www.univ-ubs.fr/valoria/Orcade/RASC`

2 Overview of the JAMUS Platform

The general architecture of the platform is shown in Figure 1. The platform is implemented over RAJE, a *Resource-Aware Java Environment* whose development is also carried out in our laboratory.

2.1 Middleware support for resource monitoring and control

RAJE can be perceived as an extension of the traditional runtime environment of the Java 2 platform. This extension provides various facilities for monitoring the resources of a hardware platform, as well as for monitoring and controlling the resources used by Java application programs at runtime. It includes classes that reify system resources (CPU, network interfaces, etc.) as well as classes that model conceptual resources (sockets, files, etc.). Some of these classes are specific to RAJE, while others simply provide alternative implementations for classes of the standard Java API. For example, standard Java classes *Socket* and *File* are given specific implementations in RAJE, so that any access to the conceptual resources they model can be monitored and controlled at runtime.

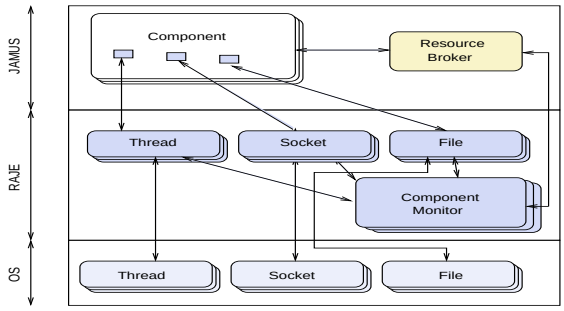

Figure 1. Overview of the platform's architecture

Most of the code included in RAJE is pure Java and, as such, is readily portable. However, part of this code consists of C functions that permit the extraction of information from the underlying OS, and the interaction with inner parts of the JVM (Java Virtual Machine). To date RAJE is implemented under Linux, and the JVM it relies on is a variant of TransVirtual Technology's Kaffe 1.0.6. System resources are monitored by polling various files in the */proc* pseudo file-system of the Linux OS. The JVM is modified in such a way that Java threads are implemented as standard Linux native threads. This approach gives us better control over the CPU resource. For example, it makes it possible to monitor the amount of CPU consumed by each Java thread, since this information is readily available under */proc* for any Linux thread.

RAJE is dedicated to supporting high-level resource-aware applications, such as adaptive systems, security-oriented systems, or QoS-oriented systems. The JAMUS platform lies at the frontier between the two latter categories. It controls the resources used

by Java application programs in order to ensure a secure runtime environment for these programs, as well as guaranteed QoS as far as resource availability is concerned.

2.2 Resource contracting in JAMUS

As mentioned in Section 1, resource control in JAMUS is based on a contractual approach. A four-level classification of the various kinds of contracts encountered in object-oriented programming has been proposed in [4]. According to this classification, the contracts we consider in JAMUS fall into the fourth level, which is that of contracts pertaining to non-functional properties and requirements.

Whenever a software component applies for being deployed on the platform, this component must specify explicitly what resources it will need at runtime, and in what conditions. Moreover access conditions can be specified in both a qualitative way (*eg* access permissions) and a quantitative way (*eg* access quotas). By specifying its requirements regarding resource access privileges and quotas, the candidate component requests a specific service from the JAMUS platform. At the same time it promises to use no other resource than those mentioned explicitly. Likewise, when the platform accepts a candidate component it promises to provide the component with all the resources it requires. At the same time it reserves the right to sanction any component that would try to access other resources than those it required.

The main originality of our approach with respect to related works (such as those mentioned in Section 6) lies in the fact that a specific contract must be subscribed between the JAMUS platform and each software component it accommodates. Based on these contracts, JAMUS provides some level of quality of service regarding resource availability. It also provides components with a relatively safe runtime environment, since no component can access or monopolise resources to the detriment of other components.

Contract subscription. Contract subscription in JAMUS relies on a two-step procedure. Any software component that applies for being deployed on the JAMUS platform must first pass an admission control examination. The requirements of a candidate component regarding resource utilisation must be expressed as a set of so-called resource utilisation profiles. Section 3 gives a detailed description of these profiles. The requirements of a candidate component are examined by a resource broker in order to decide if this component can be admitted on the platform. Admission control is based on a resource reservation scheme: a component is admissible only if the resources it requires are available on the platform in sufficient quality and quantity. When a component is declared admissible, the resource broker reserves the required resources for this component. The role and features of the resource broker are further detailed in Section 4.

When the admission control step and the resource reservation step are complete, a candidate component can start running on the platform. However the platform and the component are bound by a contract: the platform expects that the component will not attempt to access other resources than those it required, and the component expects that the platform will provide the resources it required.

Contract monitoring. Contract monitoring would not be necessary if the components deployed on the JAMUS platform could all be considered as trustworthy. If that was the case, any component could reasonably be expected to behave exactly as promised, and to use only those resources it required. Indeed, works are in progress that aim at providing so-called ”trusted components” [15,12,14]. Hence, there is hope that, in the future, contract monitoring will not necessarily be required in all circumstances.

JAMUS is being developed in order to provide a safe and guaranteed runtime environment for components of dubious origin, such as application programs or applets downloaded from remote Internet sites. Consequently, any component deployed on the platform is considered as a potential threat throughout its execution.

Once a component has been accepted on the JAMUS platform, this does not necessarily mean that at runtime this component will respect the contract it subscribed with the platform. A component designed in a clumsy way – or in a malevolent perspective – may attempt to misbehave by accessing resources it did not explicitly ask for. In order to prevent such problems any component deployed on JAMUS is considered as non-trustworthy. Its execution is monitored so as to to check that it never attempts to access resources in a way that would violate the contract it subscribed with the platform during the admission control step. The mechanisms that permit the supervision of a component at runtime are presented in Section 5.

3 Describing Restrictions and Requirements: the Key to Contracts

At startup the JAMUS platform is given a description of the resources it can make available to hosted components, as well as instructions on how these resources should be used by components. RAJE and JAMUS together provide a series of interfaces and classes that permit to model resources and access conditions as so-called “resource utilisation profiles”. Such profiles can thus be defined as Java code, and handled at runtime as standard Java objects.

Setting restrictions on the platform. The piece of code reproduced in Figure 2 shows how resource usage profiles can be defined in order to describe and to set restrictions on the resources available on the platform. In this example the platform is provided with three distinct profiles.

An instance of class *ResourceUtilisationProfile* basically aggregates three objects, which implement the *ResourcePattern*, *ResourcePermission*, and *ResourceQuota* interfaces respectively (see Figure 3).

JAMUS provides specific implementations of these interfaces for each resource type. For example, some of the classes that implement the ResourcePattern interface are shown in Figure 4. By including a given type of *ResourcePattern* in a *ResourceUtilisationProfile* one indicates that this profile is only relevant for those resources whose characteristics match the pattern, and that the *ResourcePermission* and *ResourceQuota* objects defined in this profile only pertain to this particular set of resources.

For example, the *SocketPattern* defined in profile *C1* (see Figure 2) specifies that this profile only concerns socket resources. The access permissions and transmission quotas defined in *C1* should thus be enforced only on sockets. The *FilePattern* defined

```
int MB = 1024*1024;
ResourceUtilisationProfile C1, C2, C3;

// Global restrictions set on all socket-based
// communications: 200 MB sent, 500 MB received.
C1 = new ResourceUtilisationProfile(
        new SocketPattern(),
        new SocketPermission(SocketPermission.all),
        new SocketQuota(200*MB, 500*MB));

// Global restrictions set on any access
// to directory /tmp: 100 MB written, 100 MB read.
C2 = new ResourceUtilisationProfile(
        new FilePattern("/tmp"),
        new FilePermission(FilePermission.all),
        new FileQuota(100*MB, 100*MB));

// Selective restriction set on any access
// to directory /tmp/jamus: 15 MB written, 40 MB read.
C3 = new ResourceUtilisationProfile(
        new FilePattern("/tmp/jamus"),
        new FilePermission(FilePermission.all),
        new FileQuota(15*MB, 40*MB));
```

Figure 2. Example of restrictions imposed on the JAMUS platform.

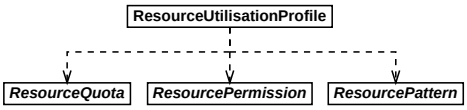

Figure 3. Object-oriented modelling of resource utilisation profiles.

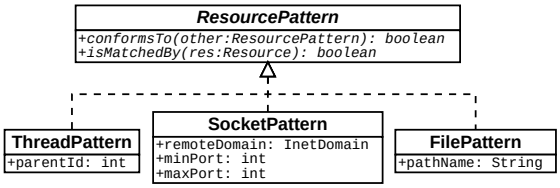

Figure 4. Excerpt from the resource pattern hierarchy, as defined in JAMUS.

in profile *C2* specifies that this profile concerns only file resources, and more precisely those files that are located under the */tmp* directory. The permissions and quotas defined in *C2* should thus be enforced only on runtime components attempting to accessing files in */tmp*. Profile *C3* sets further restrictions on directory */tmp/jamus*, imposing smaller quota values on this specific directory.

Specifying hosted components' requirements. Whenever a candidate component is submitted to the admission control step, it must specify its own requirements with respect to the resources offered by the platform. These requirements can also be modelled as instances of the class *ResourceUtilisationProfile*.

Figure 5 shows how a component can specify its requirements by defining the static method *getResourceRequirements()*, which returns a set of *ResourceUtilisationProfiles*. This method is systematically called by the platform's launcher in order to ask candidate programs for their requirements.

Checking requirements against restrictions. The interfaces *ResourcePattern*, *ResourcePermission* and *ResourceQuota* all define a boolean function *conformsTo()*, which can be used to check the conformity between two objects that implement the same interface (as shown in Figure 4 in the case of interface *ResourcePattern*). Conformity between patterns, permissions, and quotas is defined as follows:

- A pattern *Sa* is said to conform to another pattern *Sb* if any resource object whose characteristics satisfy the selection criterion defined in *Sa* also satisfy that of *Sb*.
- A permission *Pa* is said to conform to another permission *Pb* if any operation allowed in *Pa* is also allowed in *Pb*.
- A quota *Qa* is said to conform to another quota *Qb* if any amount (or rate) specified in *Qa* is smaller than (or equal to) that specified in *Qb*.

The function *conformsTo()* can thus be used at runtime to check a program's requirements against the restrictions imposed on the platform. With this function one can readily identify the requirements and restrictions that concern the same category of resources, and then verify that these requirements are compatible with those restrictions.

4 Admission Control and Resource Reservation: Contract Subscription

The resource broker implemented in JAMUS is inspired from that described in [13]. It implements a reservation mechanism in order to guarantee the availability of resources for hosted components.

Initialisation of the resource broker. At startup, the broker receives the set of resource utilisation profiles that describe the restrictions imposed on the platform's resources. This information is used by the broker to build its own "perception" of the resources initially available on the platform, and of the conditions set on any access to these resources.

```
public class MyProgram {
  public static Set getResourceRequirements(String[] args) {
    int MB = 1024*1024;
    ResourceUtilisationProfile R1, R2, R3, R4;

    // Global requirement for all socket-based
    // communications: 20 MB sent, 80 MB received.
    R1 = new ResourceUtilisationProfile(
                    new SocketPattern(),
                    new SocketPermission(SocketPermission.all),
                    new SocketQuota(20*MB, 80*MB));

    // Selective requirement for connections to the specified
    // Internet domain: 5 MB sent, 12 MB received.
    R2 = new ResourceUtilisationProfile(
                     new SocketPattern("univ-ubs.fr"),
                     new SocketPermission(SocketPermission.all),
                     new SocketQuota(5*MB, 12*MB));

    // Global requirement for the file system: access limited
    // to directory '/tmp': 30 MB written, 40 MB read.
    R3 = new ResourceUtilisationProfile(
                     new FilePattern("/tmp"),
                     new FilePermission(FilePermission.all),
                     new FileQuota(30*MB, 40*MB));

    // Selective requirement concerning access to directory
    // /tmp/jamus/data: 5 MB read, write access not required.
    R4 = new ResourceUtilisationProfile(
                     new FilePattern("/tmp/jamus/data"),
                     new FilePermission(FilePermission.readOnly),
                     new FileQuota(0, 5*MB));

    Set req = new HashSet();
    req.add(R1); req.add(R2); req.add(R3); req.add(R4);
    return req;
  }

  public static void main(String[] args) { . . .  }:
}
```

Figure 5. Example of an simple application program that specifies its own requirements regarding the resources it plans to use at runtime.

Admission of a candidate component. Whenever a candidate component is submitted to the admission control test, the resource broker examines the requirements of this component. For each basic requirement, the broker must decide if this requirement is admissible. A requirement is declared admissible if it violates none of the restrictions imposed on the platform. The candidate component itself is declared admissible if all its requirements are admissible. A component can be hosted on the JAMUS platform only after it has been declared admissible by the resource broker.

Consider the requirements *R2* and *R3* expressed by the candidate component reproduced in Figure 5, and assume that the broker is checking these requirements against the platform's restrictions defined in Figure 2. Since the pattern defined in *R2* only conforms to that of *C1*, the permission and quota conditions required in *R2* should only be checked against those imposed in *C1*. The broker must thus check that the access permissions (resp. quotas) required in *R2* do not violate the restrictions imposed in *C1*. In the present case, *R2* can be declared admissible by the broker.

The pattern defined in *R3* conforms to those defined in both *C2* and *C3*. Access to the part of the file-system considered in *R3* should thus be performed according to the restrictions imposed in both *C2* and *C3*. Analysis of the access permissions required in *R3* shows that they contradict neither the restrictions imposed in *C2*, nor those imposed in *C3*. On the other hand, although the access quotas required in *R3* are compatible with those imposed in *C*2, they do not conform to those imposed in *C*3. Requirement *R3* should thus be declared as non-admissible by the resource broker: the requirement expressed in *R3* could not be satisfied by JAMUS without violating at least one of the restrictions imposed on the platform. As a consequence, the candidate component shown in Figure 5 should be declared as non-admissible by the platform, assuming that this platform is bound by the restrictions shown in Figure 2.

Resource reservation and release. When a candidate component has successfully passed the admission control test, it can start running on the platform. However, JAMUS commits itself to providing hosted components with the resources they require. The resource broker is responsible for this commitment on behalf of the entire platform. Resource reservation is achieved by dynamically updating the broker's "perception" of the resources available on the platform.

Once a component has been declared admissible by the broker, and before this component actually starts running, the broker updates the quota values in the platform's restrictions, based on the component's requirements. Hence, whenever the requirements of a new candidate component are examined by the broker, these requirements are actually checked against a set of profiles that model the currently available resources, rather than the resources that were available when the broker was first initialised.

Likewise, when a component reaches completion the broker updates the quota values in the platform's restrictions accordingly, so as to account for the release of the resources that were reserved for this component.

5 Resource Access Supervision: Contract Monitoring

5.1 Monitoring entities

Every component hosted by the JAMUS platform runs under the control of a component monitor. This monitor uses the resource utilisation profiles provided by the component to instantiate as many resource monitors.

A resource monitor admits a *ResourceUtilisationProfile* as a construction parameter. Its mission is to monitor the resources whose characteristics match the pattern defined in this profile, and to enforce the profile's access permissions and quotas on these resources.

JAMUS includes different kinds of resource monitors. For example, the class *SocketMonitor* implements a resource monitor dedicated to socket resources. When monitoring the resources used by a hosted component, the role of a *SocketMonitor* is to check that the socket resources that satisfy the selection criterion defined in its *SocketPattern* are used according to the conditions specified in its *SocketPermission* and *SocketQuota*.

As an example, consider the component shown in Figure 5, and assume that this component has been admitted by the JAMUS platform. When loading this component, the platform creates a component monitor. This monitor examines the requirements of the component, and since these requirements are expressed as four resource utilisation profiles, it creates four resource monitors. Two of these resource monitors are *SocketMonitors*, and the other two are *FileMonitors*. The first *SocketMonitor* receives profile *R1* as a construction parameter. From this time on it is thus dedicated to monitoring the use of all the *Socket* resources the hosted component may create at runtime, while enforcing the global quotas specified in *R1* (no more than 20 MBytes sent, no more than 80 MBytes received) on these sockets. The second *SocketMonitor* receives *R2* as a construction parameter: it will thus have to monitor only sockets connected to hosts of the remote Internet domain *"univ-ubs.fr"*, enforcing the quotas specified in *R2* (no more than 5 MBytes sent, no more than 12 MBytes received) on these sockets.

Notice that at runtime any socket monitored by the second *SocketMonitor* shall be also monitored by the first one, as its characteristics will match the patterns defined in both *R1* and *R2*. A resource may thus be supervised by several monitors, just like a single monitor may have to supervise several resources simultaneously.

The remaining of this section gives an overview of the mechanisms component monitors and resource monitors rely on.

5.2 Resource tracking

In JAMUS all resources are modelled as Java objects (instances of classes *Socket*, *File*, *CPU*, etc.), which can be created and destroyed (or, more precisely, de-referenced) dynamically by a hosted component. As a consequence, the component monitor that is responsible for monitoring a given hosted component must be informed about any creation or destruction of a resource object by this component. The JAMUS platform implements a resource register, whose role is to identify and to keep track of all resource objects used by a hosted component.

In the current implementation of the platform a distinct resource register is associated with each hosted component. Moreover, each component is loaded using a distinct *ClassLoader*. This approach ensures that resource objects used by different components are registered separately. It also guarantees that two hosted components do not share the same name-space, hence preventing resource capture and corruption between concurrent components.

5.3 Monitoring models

Resource monitoring is said to be achieved synchronously when any attempt to access a given resource can be intercepted and checked immediately by the monitors associated with this kind of resource. Synchronous monitoring can be obtained quite easily when a resource is modelled by an accessor object, that is, when accessing a resource comes down to calling a method on a Java object that represents this resource in the Java environment. Classes *Socket*, *DatagramSocket*, *File*, etc. are accessor classes: they model conceptual resources, and as such they define accessor objects that can be submitted to synchronous monitoring.

All resources cannot be monitored using the synchronous approach, though. For example, although all Java components (or, more precisely, all Java threads) consume shares of the CPU resource, they do not do so explicitly by calling methods on an object that would model the underlying hardware CPU. Instead, access to the CPU is controlled exclusively by the scheduler of the Java Virtual Machine, or by that of the underlying operating system [18]. In order to deal with resources that cannot be monitored synchronously, such as the CPU, we propose to do with asynchronous monitoring. Monitoring a resource asynchronously consists in consulting the state of this resource every now and then, in such a way that the time of the observation does not necessarily coincide with the time of an attempt to use the resource.

RAJE provides abstractions and implementation alternatives for performing both kinds of monitoring. Some of these facilities are used in JAMUS to put resources under the control of resource monitors.

Implementation of the synchronous model. Synchronous monitoring is obtained in RAJE by implementing a call-back mechanism in resource classes. Any resource object admits a set of listeners. Whenever a method is called on a resource object by an application component, the resource object informs all its registered listeners that an attempt is being made to access the resource it models. A listener can refuse that a certain operation be performed on a resource by returning a *ResourceAccessException* signal to the corresponding resource object. In such a case the resource object must abort the operation considered, and return the appropriate exception signal to the application component.

With this approach a resource monitor that implements the synchronous model can keep informed about any operation attempted on the resource objects it monitors.

Implementation of the asynchronous model. In order to be observable asynchronously, an object that models a resource must be able to produce an observation report

on demand. In RAJE, an observation report is an object that implements the *ObservationReport* interface. RAJE provides a specific implementation of this interface for each type of resource considered to date. Moreover, each resource class defines a method *observe()* that returns the appropriate kind of *ObservationReport*.

For example, a *Thread* object (threads are considered as resources in JAMUS) can produce a *ThreadReport*, which reports the current priority level of this thread, as well as the amount of CPU it consumed during the last observation period. Likewise, the method *observe()* implemented in class *Socket* returns a *SocketReport*, which reports the current characteristics of the corresponding socket (remote address and port, number of bytes sent and received since this socket was opened, etc.).

RAJE also provides mechanisms for controlling resources. For example many resource classes defined in RAJE implement interface *Lockable*. When a resource object is locked, utilisation of this object by an application component is denied, and any attempt to access the associated resource raises an exception signal.

Notice that observation and locking facilities have been kept separate in RAJE. The reason for this separation is that although all resources can be observed quite easily, there are resources (such as the system CPU) that can hardly be locked.

5.4 Dealing with Faulty Components

Resource monitors in JAMUS rely on either the asynchronous or the synchronous facilities implemented in RAJE in order to monitor the resources whose characteristics match their pattern. Each monitor enforces the access permissions and quotas defined in its profile on the resource objects it monitors. Whenever a transgression is observed, the monitor notifies the resource broker. The broker then terminates the faulty component. Release of the resources this component used is accounted by the broker, so that these resources can later be reassigned to new candidate components.

6 Related Work

The Java Runtime Environment (JRE) implements the so-called *sandbox* security model. In the first versions of the JRE, this security model gave local code –considered as safe code– full access to system resources, while code downloaded from the Internet (for example under the form of an applet) was considered as untrusted, and was therefore only granted access to a limited subset of resources [9]. With the Java 2 platform this restrictive security model was abandoned for a new model that relies on the concept of protection domain [9,10,18]. A protection domain is a runtime environment whose security policy can be specified as a set of permissions. An access controller is associated with each protection domain. It checks any resource access performed from this domain, and it implements the security policy as defined by the permissions associated with the domain.

J-Kernel extends this approach by permitting communication and data sharing between protection domains [11]. Communication between domains is however limited to method calls on so-called capability objects.

The security models implemented in J-Kernel and in the JRE rely on stateless mechanisms. Access to a specific resource cannot be conditioned by whether the very same resource was accessed previously, or by how much of this resource was consumed previously. Hence, quantitative constraints (amount of CPU, I/O quotas, etc.) cannot be set on the resources accessed from protection domains. As a consequence, the security mechanisms implemented in J-Kernel and in the JRE cannot prevent abnormal operations resulting from an abusive consumption of resources (denial of service attacks, etc.).

Environments such as JRes [3,5], GVM [2], and KaffeOS [1] partially solve the above-mentioned problem. They include mechanisms that permit to count and to limit the amounts of resources used by an active entity (a thread for JRes, a process for GVM and KaffeOS). However, resource accounting is only achieved at coarse grain. For example it is possible to count the number of bytes sent and received by a thread (or by a process) through the network, but it is not possible to count the number of bytes exchanged with a given remote host, or with a specific remote port number. Similarly, it is possible to count how many bytes have been written to (or read from) the file system, but it is not possible to set particular constraints on the use of specific directories or files.

Projects Naccio [7,6] and Ariel [16] are dedicated to security. Both provide a language for defining a security policy, as well as mechanisms for enforcing this policy while an application program is running. Security policy enforcement is carried out statically, by rewriting the application program byte-code as well as that of the Java API. An advantage of this approach is that the cost of the supervision of a program is kept at a minimum, for 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. Unlike the JAMUS platform, though, Naccio and Ariel can hardly ensure the supervision of an application program, when the security policy must be derived from the resource requirements specified by the application program itself.

[8] presents a resource-constrained sandboxing approach that shows much similitude with ours. The sandbox described in this paper can enforce qualitative and quantitative restrictions on the system resources used by application programs, while providing these programs with soft guarantees of and fairness of resources. It monitors the application's interactions with the underlying operating system, pro-actively controlling these interactions in order to enforce the desired behaviour. The paper distinguishes between implicit and explicit requests to the OS, and it shows that resource utilisation can be constrained either by intercepting explicit requests to the OS, or by monitoring the frequency of implicit requests. This distinction obviously compares with our observation that resource monitoring should be performed either synchronously or asynchronously, depending on the kind of resource considered, and on the desired level of control. Unlike the other works reported in this section, and unlike JAMUS, the sandbox described in [8] is not object-oriented. It relies on standard shared system libraries available in Windows NT and in Linux, in order to provide sandboxing for executable programs.

7 Discussion and Perspectives

The development of the JAMUS platform (and of the underlying environment RAJE) is still in progress. As a consequence, JAMUS is obviously still limited and quite perfectible. In the following we list some of the problems we should address shortly, as well as some of the lines we plan to work along in the future.

In the current implementation, resource usage profiles must be expressed as Java source code (as shown in Section 3). This is a rather crude, and sometimes not-so-convenient creation procedure. We should soon raise this limitation, though, as it would be most interesting if profile objects could be stored in files, and retrieved from these files at startup. An approach to achieve this goal would be that all *ResourceUtilisationProfile* objects implement the standard Java interface *Externalizable*. Another interesting approach would be to define a dedicated XML-based language, so that profiles can be defined and edited using some of the tools developed around the XML technology.

The current syntax, grammar, and semantics associated with resource utilisation profiles have not been formerly defined. Besides, they could certainly be improved so as to gain expressiveness and flexibility. For example it would be interesting if resources could be reserved for groups of components, rather than for individual components. Another possibility would be to complement the mechanism of resource reservation with alternative access schemes (such as best effort, priority-based access, etc.).

Resource contracting in JAMUS relies on a quite simplistic model: the requirements of a component must all be known and fully expressed *before* this component can be admitted on the platform. In many cases, though, the requirements of a component can hardly be defined statically, because they depend on parameters the component should only discover at runtime, or because they are likely to change dynamically while the component is running. The architecture of JAMUS gives provision for more flexible contract-management mechanisms. New facilities should be included in the platform in the near future so as to permit that contracts be subscripted, cancelled, or modified dynamically throughout a component's lifetime.

Since the development of the platform is not complete, no performance evaluation has been performed so far. However, the fact that JAMUS relies on fully dynamic mechanisms suggests that the cost of monitoring Java components at runtime might be rather high. In compensation, we believe that this cost could eventually be considered as acceptable, considering that the platform makes it possible to control resources at a very fine grain, and considering it should soon be possible for a component to negotiate contracts dynamically.

JAMUS constitutes a demonstrator platform, with which we experiment with the idea of resource-constrained software deployment. 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 *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. Nataraj Bagaratnan and Steven B. Byrne. Resource Access Control for an Internet UserAgent. In *Third USENIX Conference on Object-Oriented Technologie and Systems*, 1997.
4. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract-aware. In IEEE, editor, *Computer*, page 38 44. IEEE, June 1999.
5. Grzegorz Czajkowski and Thorsten von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.
6. David Evans. *Policy-Directed Code Safety*. PhD thesis, Massachussets Institute of Technology, February 2000.
7. David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy*, May 1999.
8. Fangzhe Chang and Ayal Itzkovitz and Vijay Karamcheti. User-level Resource-constrained Sandboxing. In -. 4 th USENIX Windows Systems Symposium, August 2000. qualitative and quantitative restrictions on CPU, network and memory. This sandbox model is implemented under WindowsNT and Linux.
9. Li Gong. Java Security: Present and Near Future. *IEEE Micro*, -:14–19, May 1997.
10. 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.
11. Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *USENIX Annual Technical Conference*, June 1998.
12. J.-M. Jézéquel, D. Deveaux, and Y. Le Traon. Reliable Objects: a Lightweight Approach applied to Java. *IEEE Software*, 18(4):76–83, July/August 2001.
13. Kihun Kim and Klara Nahrstedt. A Resource Broker Model with Integrated Reservation Scheme. In *IEEE International Conference on Multimedia and Expo (II)*, pages 859–862, 2000.
14. U. Lindqvist, T. Olovsson, , and E. Jonsson. An Analysis of a Secure System Based on Trusted Components. In *Proc. 11th Ann. Conf. Computer Assurance*, pages 213–223. IEEE Press, 1996.
15. B. Meyer, C. Mingins, and H. Schmidt. Providing Trusted Components to the Industry. *IEEE Computer*, pages 104–15, May 1998.
16. 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.
17. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, 1998.
18. Bill Veners. *Inside the Java 2 Virtual Machine*. Mac Graw-Hill, 1998.