JAMUS: Java Accommodation of Mobile Untrusted Software

Nicolas Le Sommer, Frédéric Guidec

JAMUS: Java Accommodation of Mobile Untrusted Software

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

1 Introduction

Security is a major issue for mobile components that roam the Internet. When downloading a software component from the Internet, it is often impossible to decide in advance if this piece of code should be considered as safe or potentially dangerous for the local system. A malicious – or simply buggy – component might put the whole system in jeopardy, as it might destroy crucial data files, or consume too much CPU time, memory, or network bandwidth. Another important issue when hosting mobile components is resource management. Some components can do very well with sparse resources, while others require predictable or guaranteed levels of quality of service (QoS) regarding resource availability.

With the JAMUS *(Java Accommodation of Mobile Untrusted Software)* platform we tackle these problems based on a contractual approach of resource management and access control. JAMUS can accommodate mobile Java 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).

Nowadays, terms such as "mobility" and "mobile component" admit many definitions in the computer science community. Many projects are in progress that aim at supporting the mobility of software components (including mobile agents), as well as the design and the deployment of mobile applications [KT98, Bou01, Wel01, GCKR00]. The work reported in this paper does not focus on component mobility as such. The mobile components we consider are simply untrusted Java programs and applets, which could be downloaded from remote Internet sites or received as Email attachments. JAMUS is basically an experimental platform, dedicated to offering a safe and guaranteed runtime environment for such basic mobile components.

The remaining of this paper is organised as follows. Section 2 lists recent works that aim at gaining control of the resources used by application programs. The general architecture of the JAMUS platform is discussed in Section 3. This section also gives an overview of RAJE, the runtime environment JAMUS relies on. Section 4 introduces the notion of resource utilisation profile. 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. Most notably, they are the keys to resource management and control, which are under the responsibility of a resource broker. The role and the main features of this broker are detailed in Section 5. Section 6 shows how hosted components are constantly monitored while running on the platform, and it details the mechanisms that permit this monitoring. Section 7 concludes this paper.

2 Resource Control in Related Work

The problem of monitoring and controlling the resources used by runtime entities (ie programs, processes, or threads) has been addressed in several projects. This section lists some of the most recent ones, focusing on those that specifically consider the issues related to resource control in a Java environment.

The Java Runtime Environment (JRE) of the Java 2 platform implements a security model that relies on the concept of protection domain [Gon97, GS98, Ven98]. 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 [HCC+98].

The security models implemented in J-Kernel and in the

JRE rely on stateless mechanisms. Access conditions can thus be expressed in qualitative terms, but not in quantitative terms (amount of CPU, I/O quotas, etc.).

Environments such as JRes [BB97, CvE98], GVM [BTS+00], and KaffeOS [BHL00] 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.

Projects Naccio [ET99, Eva00] and Ariel [PH99] permit the control of resource consumption at a very fine grain. Both provide a language for defining a precise 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. Generating an API dedicated to a specific security policy is thus a quite expensive procedure. Consequently, the approach proposed in Naccio and Ariel is mostly dedicated to the generation of predefined Java APIs that each enforce a generic security policy.

[CIK00] presents another interesting resource-constrained sandboxing approach. 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 work presented in [CIK00] differs from those mentioned previously, as it does not specifically address the control of the resources consumed by Java programs. Instead it relies on the standard shared system libraries available in Windows NT and in Linux, in order to provide sandboxing facilities for traditional executable programs.

Our objective is to offer a safe and guaranteed runtime environment for mobile components. This objective implies that resource consumption be kept under control. As a consequence, our work shows similitude with the above-mentioned projects. However, while working on the design and implementation of the JAMUS platform we try to keep the following aims in mind.

- *Monitoring and control of resource utilisation should be feasible at a very fine grain.*
  It should be possible to monitor the access to a network interface or to the file system, but it should also be possible to monitor the access to specific hosts and services through the network interface, or to specific directories and files in the file system.
- *Access conditions should be expressible in qualitative and quantitative terms.*
  It should be possible to set access permissions on the resources used by Java components, but it should also be possible to set access quotas on these resources.
- *The security policy enforced on a component should be derived from the requirements expressed by this component.*
  Each Java component hosted by the platform should run in a specific environment, whose "dimensions" should be deduced from the component's requirements.

3 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.

Resource-Aware Java Environment. RAJE can be perceived as an extension of the traditional runtime environment of the Java 2 platform. This extension 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.

RAJE thus provides various facilities for monitoring and controlling the resources used by Java application programs. System resources can be monitored as well, although JAMUS does not take advantage of this possibility yet.

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

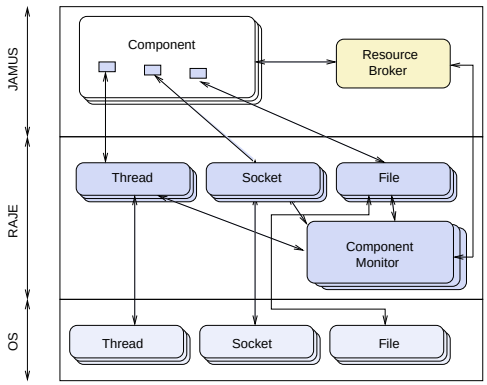

Figure 1: Overview of the platform's architecture

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.

Contractual Approach of Resource Control. Resource control in JAMUS is based on a contractual approach. Whenever an application program applies for being hosted by the platform, this program 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 program 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 program it promises to provide the program with all the resources it requires. At the same time it reserves the right to sanction any program that would try to access other resources than those it required.

The main originality of this approach lies in the contracts that must be signed between the JAMUS platform and the application components it accommodates. Based on these contracts, JAMUS provides some level of quality of service regarding resource availability. It also provides hosted programs with a relatively safe runtime environment, since no hosted program can access or monopolize resources to the detriment of other hosted programs. These characteristics both result from the application of two main principles, as discussed below.

Admission Control. Any application program that applies for being hosted by the JAMUS platform must pass an admission control examination. This step is under the responsibility of a resource broker. The broker interacts with the application program in order to collect its requirements regarding resources. It then uses this information in order to decide if the candidate program can be accepted on the platform.

Resource requirements must be expressed as a set of so-called resource utilisation profiles. Section 4 gives a detailed description of these profiles, while the resource broker is further discussed in Section 5.

Systematic Program Monitoring. Once an application program has been accepted for running on the JAMUS platform, this does not necessarily mean that at runtime this program will behave exactly as expected by the platform. A hosted program should never attempt to use other resources than those it required during the admission control step. Yet, a program 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 program hosted by 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 signed with the platform during the admission control step. The mechanisms that permit the supervision of a program at runtime are presented in Section 6.

4 Resource Utilisation Profiles

At startup the JAMUS platform is given a description of the resources it can make available to hosted application programs. This information consists in an enumeration of all – or part of – the resources that are available on the local system, together with indications on how to use these resources.

Whenever a candidate application program is submitted to the admission control step, this program must specify its own requirements with respect to the resources offered by the platform. The information provided by the program consists in an enumeration of the resources it plans to use at runtime, together with indications on how it plans to use these resources.

There is an obvious similitude between the information given to the platform at startup, and that provided by candidate programs at admission time. The concept of resource utilisation profile was introduced in order to capture this similitude, while providing a means to handle both kinds of information at runtime.

Describing Resource Utilisation Profiles. From an application program's point of view, a resource utilisation profile describes a specific requirement of the program with respect to the resources offered by the platform. RAJE provides a class *ResourceUtilisationProfile* that models such a requirement. Resource utilisation profiles can thus be defined as Java code, and handled at runtime as standard Java objects.

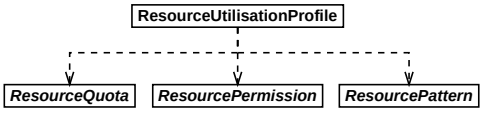

Figure 2: Object-oriented modelling of resource utilisation profiles.

An instance of class *ResourceUtilisationProfile* basically aggregates three objects, which implement the *ResourcePattern*, *ResourcePermission*, and *ResourceQuota* interfaces respectively, as shown in Figure 2. These objects are referred to through attributes *pattern*, *permission*, and *quota* in a *ResourceUtilisationProfile* object.

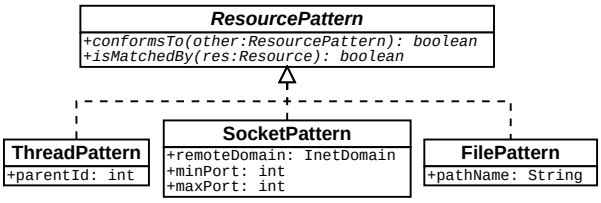

Figure 3: Excerpt from the resource pattern hierarchy, as defined in JAMUS.

The attribute *pattern* in a *ResourceUtilisationProfile* hence refers to a *ResourcePattern* object. The role of this attribute is to identify precisely the kind of resource considered in the profile. Some of the resource patterns implemented in JAMUS are shown in Figure 3. This set is neither complete, nor exhaustive. New patterns could be included in JAMUS in the future, as new kinds of resources are allowed for, or as new modes of utilisation are admitted for known resource types.

By inserting the desired pattern in a resource utilisation profile, one can create a profile that concerns only a subset of the resources available on the JAMUS platform. For example, the following instruction,

```
new SocketPattern("univ-ubs.fr", 0, 1023);
```

creates a resource pattern that considers only socket resources, and more precisely those sockets that connect the local system to a specific set of remote hosts and ports. In this example the remote host must belong to the Internet domain *"univ-ubs.fr"*, and the remote port must be in the specified port range.

A *ResourcePattern* object can thus be used to describe a set of resources. By including a given pattern in a *ResourceUtilisationProfile* one indicates that this profile only concerns those resource objects whose features match the pattern. Consequently, a *ResourcePattern* object also plays a role at runtime, as it can be used to select between a set of resource objects those objects whose access should be checked against the permission and quota associated with the utilisation profile considered. To achieve this goal the interface *ResourcePattern* defines a boolean function *isMatchedBy()*. This function takes a resource object as a parameter, and it returns a boolean value that indicates if this object satisfies a given selection criterion. Each class that implements interface *ResourcePattern* thus defines a specific selection policy. In the simplest case, such a class may select object resources based on their type. For example, the class *SocketPattern* may implement the function *isMatchedBy()* in such a way that it simply returns true for *Socket* objects, and false for any other type of resource object. However, such a simplistic filter could only distinguish socket resources from other kinds of resources. Resource patterns can also implement more elaborate selection criteria. For example the class *SocketPattern* shown in Figure 3 admits an Internet domain and a port range as construction parameters. For this class, the selection criterion implemented in function *isMatchedBy()* can examine the values of the remote address and remote port of a *Socket* object in order to select specific sockets based on the remote end of a socket connection. With such a filter it is possible to select only those sockets that connect the local system to a specific remote domain, and to specific remote ports in this domain.

Attributes *permission* and *quota* in a *ResourceUtilisationProfile* object both specify utilisation conditions that should apply to the resources selected by the *pattern* attribute. The *ResourcePermission* object associated with

the attribute *permission* defines these modalities qualitatively, whereas the *ResourceQuota* object associated with the attribute *quota* defines these modalities quantitatively. For example, a class *SocketPermission* can be used to set access permissions on socket resources, whereas a class *SocketQuota* can limit the number of bytes that can be sent and received through these socket resources.

Setting Restrictions on the Platform's Resources. At startup the JAMUS platform receives a set of resource utilisation profiles, which specify what resources can be made available to hosted programs, and under what conditions (access permissions and quotas). These profiles implicitly define a security policy, as they impose restrictions on the resources offered to hosted programs.

The piece of code reproduced in Figure 4 shows how such restrictions can be expressed as resource utilisation profiles. The code in this example defines three distinct profiles, hence three basic restrictions on resource utilisation. These restrictions are identified as *C1*, *C2*, and *C3* (where prefix *C* stands for "Constraint") in Figure 4.

```
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 4: Example of restrictions imposed on the JAMUS platform.

C1 applies to all socket-based transmissions, since it includes a *SocketPattern* with no specification of a remote domain or port range. It specifies that the platform is assigned a limited quota of 200 MBytes in send mode, and 500 MBytes in receive mode. *C2* and *C3* set access conditions on the file system. *C2* specifies that access is granted under directory */tmp*, with quantitative limitations set to 100 MBytes in both read and write modes. *C3* sets further restrictions on access to directory */tmp/jamus*, allowing only 15 MBytes in write mode and 40 MBytes in read mode.

Specifying Hosted Programs' Requirements. The class *ResourceUtilisationProfile* is also used by candidate application programs to express their requirements during the admission control step. In that case, an instance of *ResourceUtilisationProfile* returned by a candidate program specifies what kind of resources the program will need to access while running on the platform, and what permissions and quotas it will require for these particular 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/jamus"),
                new FilePermission(FilePermission.all),
                new FileQuota(30*MB, 40*MB));

    // Selective requirement concerning accesses 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 application program that specifies its own requirements regarding the resources it plans to use at runtime.

Figure 5 shows how a candidate application program can specify its own requirements regarding the resources offered by the JAMUS platform. In this example the

program defines a method *getResourceRequirements()*. This method is systematically called by the platform's launcher in order to ask candidate programs for their requirements.

In the present case, the requirements of the program split up into four distinct resource utilisation profiles, which are denoted as *R1*, *R2*, *R*3, and *R4* (where prefix *R* stands for "Requirement") in the figure. *R1* and *R2* concern the use of socket resources. *R1* specifies that the program requires to be allowed to send up to 20 MBytes, and to receive up to 80 MBytes via network sockets. *R2* gives further requirements regarding socket-based connections established with a particular Internet domain.

R3 and *R4* express requirements concerning the use of the file system. *R3* indicates that the program requires to be able to access directory */tmp/jamus* with read and write privileges. Moreover it may have to write up to 30 MBytes to this directory, and read up to 40 MBytes from this directory. Profile *R4* gives further requirements concerning the access to directory */tmp/jamus/data*. It specifies that at runtime the program may attempt to read up to 5 MBytes from this directory. Notice that in this case, the program does not require to be able to write to directory */tmp/jamus/data*.

Checking Requirements against Restrictions. The interfaces *ResourcePattern*, *ResourcePermission* and *ResourceQuota* all define a boolean function *conformsTo()*. This function checks the conformity between two objects that implement the same interface. For example, consider the restriction *C1* imposed on the platform in Figure 4, and the requirement *R1* expressed by a hosted program in Figure 5. The following Java statement...

```
R1.pattern.conformsTo(C1.pattern);
```

checks that the resource pattern associated with profile *R1* conforms to that of profile *C1*. A pattern *Sa* is said to conform to another pattern *Sb* if any resource object that satisfies the selection criterion defined in *Sa* also satisfies that of *Sb*. In the former statement, both patterns in *R1* and *C1* are defined as simple (parameter-less) instances of class *SocketPattern*. An evaluation of the statement would thus return true. Similarly, the pattern associated with *R2* in Figure 5 would be declared as conforming to that of *C1* (since all socket connections established with domain *"univ-ubs.fr"* are, first and foremost, socket connections). Notice that when *Sa* conforms to *Sb*, this does not necessarily imply that *Sb* conforms to *Sa*. For example, *C1*'s pattern does not conform to *R2*'s pattern.

While the function *conformsTo()* declared in interface *ResourcePattern* checks the conformity between two resource patterns, the same function is also declared with similar aims in interfaces *ResourcePermission* and *ResourceQuota*. For example, if *Pa* and *Pb* denote instances of class *SocketPermission*, one can check that all operations allowed in *Pa* are also allowed in *Pb* (although *Pb* may allow operations that are denied in *Pa*). Similarly, if *Qa* and *Qb* denote instances of class *SocketQuota*, then one can easily check that the transmission quotas allowed in *Qa* are smaller than those allowed in *Qb* (so that no transmission sequence admitted by *Qa* would violate *Qb*).

5 Resource Broker

The resource broker implemented in JAMUS is inspired from that described in [KN00]. A resource broker is an intermediate resource manager between clients and a resource scheduling system. It is responsible for handling requests from clients, and for enforcing resource allocation and release on behalf of these clients.

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

Admission Control. Whenever a candidate program is submitted to the admission control test, the resource broker examines the requirements of this program. These requirements are described as a set of resource utilisation profiles, as shown in Section 4. For each basic requirement (that is, for each profile submitted by the candidate program), 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.

In the following we detail the evaluation procedure of a single requirement. Let *R* denote this requirement. In order to decide if *R* can be declared admissible, the broker must first select the platform's restrictions that pertain to the same resources as those considered in *R*. This selection comes down to using function *conformsTo()* (as declared in interface *ResourcePattern*) to compare the pattern attribute of requirement *R* with those of all the platform's restrictions. Once a subset of these restrictions has been selected, the broker checks that the access permissions and quotas specified in *R* are compatible with the permissions and quotas imposed in each

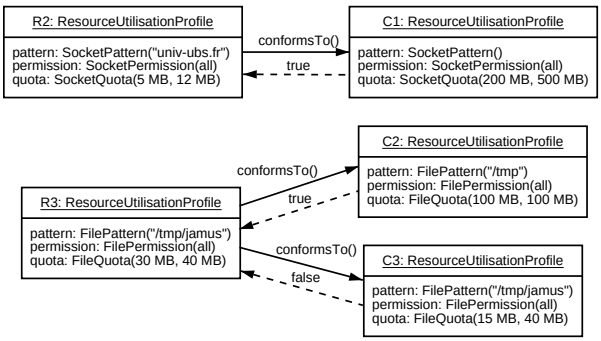

Figure 6: Checking requirements *R2* and *R3* against restrictions *C1*, *C2*, and *C3*.

of the selected restrictions. Once again the function *conformsTo()* declared in interface *ResourcePermission* (resp. *ResourceQuota)* is used to check that the permissions (resp. the quotas) specified in *R* conform to the permissions (resp. the quotas) specified in each platform's restriction.

As an example, consider the requirement denoted as *R2* in Figure 5, and assume that the broker is examining this requirement in order to decide if it can be declared admissible, allowing for the restrictions specified in Figure 4. Using function *conformsTo()* to compare the patterns of all these profiles, the broker can readily identify *C1* as the only restriction imposed on the socket resources considered in *R2*. As a consequence, all socket connections established with the Internet domain specified in *R2* should conform to the restrictions imposed in *C1*. The broker thus checks that the access permissions (resp. quotas) required in *R2* do not violate the restrictions imposed in *C1*. In the present case, *R2* requires that all access permissions be granted, which does not contradict *C1*. Likewise, *R2* requires specific transmission quotas, which are compatible with the larger quotas imposed in *C1*. After this analysis, requirement *R2* can be declared admissible by the broker (see Figure 6).

Now assume that the broker is examining requirement *R3*, as defined in Figure 5. It first observes that the pattern defined in *R3* conforms to those defined in *C2* and *C3*. In other words, access to the part of the file-system considered in *R3* should be performed according to the restrictions imposed in both *C2* and *C3*. On the one hand, analysis of the access permissions required in *R3* shows that these permissions contradict neither the restrictions imposed in *C2*, nor those imposed in *C3*. On the other hand, although the permission quotas required in *R3* are compatible with the restrictions set in *C*2, they do not conform to those set 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 program should also be declared as non-admissible by the platform.

The admission control step completes when all the program's requirements have been checked. The candidate application program is declared admissible if all its requirements are admissible. A program can be hosted on the JAMUS platform only after it has been declared admissible by the resource broker.

Resource Reservation. When a candidate program has successfully passed the admission control test, it can start running on the platform. However, JAMUS commits itself to provide hosted programs with the resources they require. Since the resource broker is responsible for this commitment on behalf of the entire platform, it implements a resource reservation scheme. The resources required by a program are reserved for this program until its execution is complete.

Resource reservation is thus achieved by updating the broker's "perception" of the resources available on the platform. This perception mostly consists in the information related to quotas in the restrictions maintained by the broker. Once a program has been declared admissible by the broker, and before this program actually starts running, the broker updates the quota values in the platform's restrictions, based on the program's requirements. The interface *ResourceQuota* defines two methods *add()* and *subtract()* that support such an "arithmetic" of quotas.

As an example, let us again consider the requirements of the program shown in Figure 5, together with the platform's restrictions reproduced in Figure 4. Assume that the quotas required in *R3* are only 10 MBytes in write mode and 20 MBytes in read mode (otherwise *R3* would not conform to *C3*, as shown in the former section). With this assumption, the program can be declared admissible by the platform's broker. Resource reservation if performed by updating the quotas specified in *C1* to *C3*, based on those specified in *R1* to *R4*. For example, since the pattern in *R3* conforms to the patterns in *C2* and *C3*, the quotas in *C2* and *C3* should be updated based on the quota required in *R3*. The *FileQuota* objects in profiles *C2* and *C3* should thus be modified by subtracting the *FileQuota* object defined in profile *R3:*

```
C2.quota.subtract(R3.quota);
C3.quota.subtract(R3.quota);
```

The broker hence updates its perception of the available resources whenever a new program is admitted on the platform. It similarly allows for the release of resources whenever a program reaches completion.

6 Runtime Monitoring

Component Monitors and Resource Monitors. When a program is loaded on the platform, the resource broker puts this program under the control of an instance of class *ComponentMonitor*. This component monitor uses the resource utilisation profiles provided by the program to instantiate as many resource monitors.

A resource monitor is an object that implements the *ResourceMonitor* interface. It takes a single *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 provides at least one type of resource monitor and one type of resource utilisation profile for each type of resource considered in RAJE. For example, the class *SocketMonitor* implements a resource monitor dedicated to socket resources. A *SocketMonitor* admits a standard *ResourceUtilisationProfile* as a construction parameter, but the attributes in this profile must refer to *SocketPattern*, *SocketPermission*, and *SocketQuota* objects respectively. When monitoring the resources used by a hosted program, the role of a *SocketMonitor* is to check that the socket resources that satisfy the selection criterion defined in the *SocketPattern* object are used according to the conditions specified in the *SocketPermission* and *SocketQuota* objects.

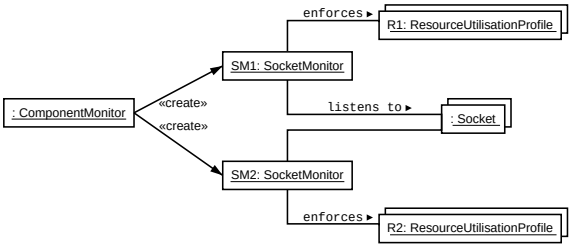

Figure 7: Creation of socket monitors SM1 and SM2 in order to enforce profiles R1 and R2 on socket resources.

As an example, consider the program shown in Figure 5, and assume that this program has been admitted by the JAMUS platform (after a slight modification on *R3*, as mentioned in Section 5). When loading this program, the platform creates a component monitor. This monitor examines the requirements of the program, and since these requirements are expressed as four resource utilisation profiles, it creates four resource monitors. Two of these resource monitors are socket monitors, and the other two are file monitors. Let us call the socket monitors *SM1* and *SM2* (see Figure 7). Monitor *SM1* receives profile *R1* as a construction parameter. From this time on *SM1* is dedicated to monitoring the use of all the *Socket* resources the hosted program 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. Monitor *SM2* 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.

The remaining of this section describes the mechanisms component monitors and resource monitors rely on.

Resource Registration and Tracking. In JAMUS all resources are modelled as Java objects (instances of classes *Socket*, *File*, *Thread*, etc.). Resource objects can be created and destroyed by a hosted application program throughout its execution. A resource register is responsible for keeping track of all these objects at runtime. This register is created as an instance of class *ResourceRegister*. All the resource classes provided in RAJE have been implemented in such a way that whenever a resource object is created or destroyed, the resource register is notified of this event (see Figure 8). By consulting the resource register, a component monitor can thus keep informed about any creation or destruction of a resource object by the program it monitors.

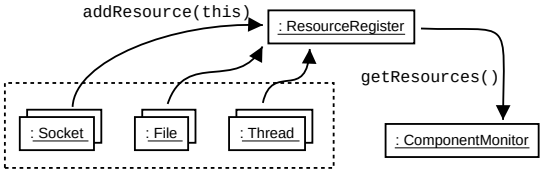

Figure 8: All resource objects notify the resource register at creation time. The component monitor consults the register to identify resources.

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

Resource Monitoring and Profile Enforcement. RAJE provides various facilities for monitoring resource objects. Some of these facilities are used in JAMUS to put resources under the control of dedicated monitors.

Resource monitoring is obtained in RAJE by implementing a call-back mechanism in resource classes. Any resource object admits a set of listeners. This set can be updated dynamically. Whenever a method is called on a resource object by an application program, 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 program.

In JAMUS, any resource monitor can register as a listener of all the resource objects whose characteristics match its pattern. With this approach a resource monitor is kept informed about any operation attempted on the resources it must monitor. Whenever such an operation is attempted, the resource monitor checks that this operation conforms to the access permissions and quotas defined in the associated resource utilisation profile. If the monitor observes that the operation considered would violate this profile, it notifies the resource broker. The broker then terminates the faulty program. It also releases the resources this program used, so that they can be reassigned to other candidate programs.

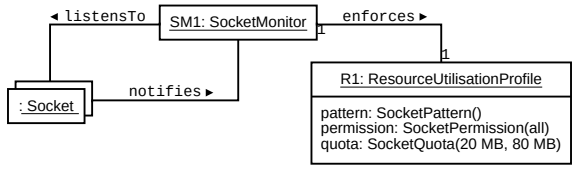

Figure 9: Socket monitor *SM1* "listens to" all socket objects, enforcing profile *R1* on these resources.

Consider the execution of the program shown in Figure 5, with the monitors *SM1* and *SM2* discussed at the beginning of this section. At runtime monitor *SM1* would register as a listener of all socket objects (see Figure 9). It could the check that all transmissions performed via these sockets conform to the permissions and quotas defined in profile *R1*. Monitor *SM2* would only have to "listen to" sockets connected to domain *"univ-ubs.fr"*, checking that transmissions via these sockets conform to the conditions defined in profile *R*2.

Notice that at runtime any socket monitored by *SM2* should be also monitored by *SM1*, as its characteristics would match the patterns defined in both *R1* and *R2*. This situation is fully justified because *SM1* needs to account for the number of bytes sent and received through all sockets, including those connected to hosts of domain *"univ-ubs.fr"*. Hence it is worth mentioning that, as a general rule, a resource may be supervised by several monitors, just like a single monitor may have to supervise several resources simultaneously.

Synchronous vs Asynchronous Monitoring. The above-mentioned approach to resource monitoring can be qualified as a synchronous approach. 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 programs (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 [Ven98].

In order to deal with resources that cannot be monitored synchronously, we plan to integrate asynchronous monitoring facilities in RAJE. Basically, monitoring a resource asynchronously comes down to 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. In the near future, RAJE should thus provide abstractions and implementation alternatives for performing both synchronous and asynchronous monitoring. Asynchronous monitoring facilities will be used in JAMUS to allow for those resources that cannot be monitored synchronously.

7 Conclusion

The JAMUS platform is dedicated to hosting mobile Java components, provided that these components can specify their requirements regarding the resources they plan to use at runtime in both qualitative and quantitative terms. JAMUS implements a contractual approach of resource management and access control in order to provide hosted components with a safe and guaranteed runtime environment. Any component that applies for running on the platform is submitted to an admission control examination. A component is admissible only if the resources it requires are available on the platform. Moreover, when a component is admitted on the platform, the resources it asked for are reserved for its sole usage. Finally, while a component is running on the platform, its execution is submitted to a constant monitoring, so that resource access violations can be readily detected and dealt with.

In its current implementation JAMUS is basically an experimental platform, whose development is still in progress. Some types of resources (including the memory) cannot be monitored yet. Moreover the platform can so far only accommodate simple mobile components such as Java application programs and applets. The accommodation of truly migrant components (such as mobile agents) could be addressed in the future.

Since the development of the platform is not complete, no performance evaluation has been performed yet. 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. Another advantage of the dynamic approach is that it makes provision for the dynamic negotiation (or re-negotiation) of resources between mobile components and the platform. This is one of the topics we plan to address in the near future.

References

[BB97] Nataraj Bagaratnan and Steven B. Byrne. Resource Access Control for an Internet UserAgent. In *Third USENIX Conference on Object-Oriented Technologie and Systems*, 1997.

[BHL00] 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.

[Bou01] Sara Bouchenak. Making Java Applications Mobile or Persistent. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, February 2001.

[BTS+00] 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.

[CIK00] Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti. User-level Resource-constrained Sandboxing. In *4'th USENIX Windows Systems Symposium*, August 2000.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.

[ET99] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy*, May 1999.

[Eva00] David Evans. *Policy-Directed Code Safety*. PhD thesis, Massachussets Institute of Technology, February 2000.

[GCKR00] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and State of the Art. Technical Report TR2000-365, Dept. of Computer Science, Dartmouth College, 2000.

[Gon97] Li Gong. Java Security: Present and Near Future. *IEEE Micro*, -:14–19, May 1997.

[GS98] 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.

[HCC+98] 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.

[KN00] 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.

[KT98] Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile-Agent Programming Systems. *IEEE Concurrency*, 6(3):52–61, /1998.

[PH99] 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.

[Ven98] Bill Veners. *Inside the Java 2 Virtual Machine*. Mac Graw-Hill, 1998.

[Wel01] Ian Welch. The Glasshouse: a Reflective Container for Mobile Code. In *7th ECOOP Workshop on Mobile Object Systems (MOS'01): Development of Robust and High Confidence Agent Applications*, June 2001.