Towards resource consumption accounting and control in Java: a practical experience

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

1 Introduction

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)*. In this project 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.

This short paper reports the design of two software products called RAJE and JAMUS, whose development is in progress in the context of project RASC. RAJE is an extension of the standard Java 2 platform. It reifies system resources as well as "conceptual" resources so that they can be observed – and, in some cases, controlled – by Java application programs. RAJE also provides facilities for implementing resource monitors, using either a synchronous or an asynchronous model. JAMUS (the acronym stands for *Java Accommodation of Mobile Untrusted Software*) is an experimental platform we develop on top of RAJE. It supports the deployment of "untrusted" 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). Emphasis is put on providing a safe and guaranteed runtime environment for such components.

2 RAJE: a *R*esource-*A*ware *J*ava *E*nvironment

Motivation

The architecture of the Java 2 platform implements a security architecture that relies on the notions of protection domain and access controller [6, 7]. A protection domain is a runtime environment whose security policy can be specified as a set of permissions. An access controller can be associated with each protection domain. Its role is to check any resource access performed from this domain. This security model relies 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 time, I/O quotas, etc.) cannot be set on the resources accessed from protection domains. As a consequence, although the security mechanisms implemented in the JRE are appropriate for achieving qualitative control over resource usage, they can hardly be used to achieve quantitative control over resource consumption.

RAJE aims at providing means to monitor both resource access and resource consumption at middleware level. Moreover it makes it possible to monitor the usage of "global" resources (CPU, system memory

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

and swap, etc.), as well as that of the resources used specifically by Java programs (TCP and UDP sockets, CPU time and memory consumed by each Java thread, etc.). RAJE can actually 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 the underlying OS and hardware platform, as well as for monitoring and controlling the resources used by Java application programs at runtime.

Resource modelling: system resources vs conceptual resources

In RAJE all resources are modelled as first-class Java objects. From this viewpoint it compares with other resource-awaenvironments, such as JRes [3] and KaffeOS [1]. RAJE includes classes that reify system resources (CPU, memory, swap, etc.) as well as classes that model "conceptual" resources, that is, resources that mostly make sense at application level (sockets, files, threads, 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* were given specific implementations in RAJE, so that any access to the conceptual resources they model can be monitored and controlled at runtime. More generally, information about any system or conceptual resource can be gathered by calling appropriate methods on the Java object modelling this resource in RAJE. Of course the nature of the information hence obtained depends on the type of resource considered. Below is a list of the types of resources RAJE currently allows for. For each resource type the information that can be collected is specified in brackets:

- System CPU (processor type, speed and cache size)
- System memory and swap (available size, current consumption)
- System processes and threads (CPU and memory consumption)
- Java threads (CPU and memory consumption for each thread or group of threads, priority level, scheduling policy)
- Java TCP and UDP sockets (local and remote addresses and ports, number of bytes sent and received, number of UDP datagrams sent and received through each socket)
- Java files (number of bytes written/read to/from each file)

RAJE was designed so as to be highly extensible. Hence new classes should be included in the distribution in the near future in order to model network interfaces (including wireless interfaces), as well as the battery of a laptop.

Extending the standard JRE: trading portability for control

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 such as the CPU and system memory are monitored by polling various files in the */proc* pseudo file-system of the Linux OS.

The JVM was modified in such a way that memory consumption can be traced on a per thread basis. Whenever a new Java object is created, the amount of memory hence consumed is put on the currently active thread's account. Likewise, whenever an object is collected by the JVM's garbage collector, the amount of memory hence freed is deducted from the account of the thread that was active when the object was created.

The JVM was also modified so 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

The standard API for Java threads was extended so that the amount of memory and CPU time (in user time and system time) consumed by any thread can be observed by simply calling a method on the thread considered. The amount of memory and CPU time consumed by a group of threads can be observed similarly.

Since Linux threads are actually POSIX threads, the API for Java threads was extended so that Java threads can take benefit of the three scheduling strategies defined in the POSIX standard (FIFO, Round-Robin, and a third strategy referred to as `SCHED_OTHER`). However in Linux the FIFO and Round-Robin strategies are reserved for processes and threads that run with superuser privileges. “Normal” processes and threads must do with the default `SCHED_OTHER` strategy, which in Linux implements a time-sharing policy with dynamic priority. All three strategies are thus available for Java threads in RAJE, but only if the JVM itself is started with superuser privileges. Otherwise all Java threads are submitted to the standard `SCHED_OTHER` policy.

Resource tracking and protection

In RAJE all resources are modelled as Java objects (instances of classes *Socket*, *File*, *Thread*, *CPU*, *Memory*, etc.). Since such objects can be created and destroyed – or, more precisely, de-referenced – dynamically by Java programs, RAJE implements a resource register, whose role is to identify and to keep track of resource objects at runtime. By consulting the resource register a program can locate all the objects that model system or conceptual resources in its name space.

With the current security model of the Java 2 architecture, several “protection domains” can be created within a single virtual machine by loading distinct application programs using as many *ClassLoader* objects. In RAJE a distinct resource register can similarly be associated with each protection domain, so that the resource objects used by different programs running in the same JVM will be registered separately. On the one hand this approach helps preventing resource capture and corruption between concurrent programs: a program will not be able to locate and gain access to the resource objects used by another program. On the other hand, some kind of collaboration between resource registers could sometimes be desirable (for example to locate and to keep track of shared resources). Although we have not tackled this problem yet, we think that it should be possible to build a hierarchy of resource registers, so that these registers can somehow collaborate and share part of their knowledge about resources in a controlled and safe way.

Support for consumption accounting: asynchronous vs synchronous monitoring

RAJE provides abstractions and implementation alternatives for performing resource consumption monitoring in either a synchronous or asynchronous way.

Resource monitoring is said to be achieved synchronously when any attempt to access a given resource can be intercepted and checked immediately. This approach can be implemented quite easily for conceptual resources, for any access to such resources implies that a method be called on a Java object (such as an instance of classes *Socket*, *DatagramSocket*, or *File*).

Synchronous monitoring is obtained by implementing a call-back mechanism in resource classes. Any resource object can admit one or several listeners. 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. This approach makes it possible to design dedicated resource monitors, that will be able to register as listeners of a specific set of resource objects in order to keep informed about the usage of these resources.

All resources cannot be monitored using the synchronous approach, though. For example, in most current implementations of the JVM Java threads are implemented as native threads. Access to the CPU is not achieved by calling methods explicitly. Instead it is controlled directly by the scheduler of the underlying operating system. In order to deal with system resources such as the CPU (system memory, network interfaces, etc.), which can hardly be monitored synchronously, we propose to do with asynchronous monitoring. Monitoring a resource asynchronously consists in consulting the state of this resource explicitly 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 order to be observable asynchronously, an object that models a resource in RAJE must be able to produce an observation report on demand. For example, a *Thread* object (threads are considered as resources in RAJE) can produce a *ThreadReport*, which reports the current priority level and scheduling policy for this thread, as well as the amount of CPU it consumed during the last observation period.

Both monitoring models are indeed complementary models. Dedicated monitors can be designed in order to trace any kind of resource usage, using either the asynchronous or synchronous facilities offered by RAJE. In any case deciding which model should be applied comes down to making a tradeoff between the precision of monitoring, and the cost of monitoring.

Indeed synchronous monitoring can be quite costly, especially when the actions being monitored require very little time (*eg* imagine that a network monitor requires to be notified synchronously whenever a single byte is being sent through a TCP socket). In return, synchronous monitoring makes it possible to monitor resource access and consumption at a very fine grain, since any access to a given resource object can be traced and accounted for.

Conversely, asynchronous monitoring permits resource objects to be polled periodically in order to consult their status. Of course, the longer the period, the less intrusive the monitoring. But in return, there is a risk that interesting or critical events be either detected too late, or even ignored because they fell in between two consecutive observations of the resources considered.

Support for resource access control: object locking

RAJE provides mechanisms for controlling the usage of resources. For example all conceptual resource classes implement interface *Lockable*. Any conceptual resource object can thus be locked and unlocked explicitly. When a resource object is locked, utilisation of this object by an application program is denied until the object is unlocked. Any attempt to access a locked resource by calling a method on the associated object raises an exception signal. This mechanism makes provision for the implementation of high-level Java-based scheduling and policy enforcement strategies. In the current implementation, though, any application program that wishes to access a locked resource object can unlock this resource on its own authority. The methods defined in interface *Lockable* should somehow be “protected” (for example using an authentication procedure) so that resource locking and unlocking can only be achieved by authorised Java components or programs.

3 Towards a contract-based approach of resource consumption

Motivation

RAJE is dedicated to supporting high-level resource-aware environments and applications, such as adaptive systems, security-oriented systems, or QoS-oriented systems. Adaptive software components could take advantage of the monitoring facilities implemented in RAJE in order to “discover” at startup the resources offered by their runtime environment, and to keep informed about the status of these resources at runtime. Security-oriented systems could rely on the same facilities to enforce security policies regarding resource access and consumption. As for QoS-oriented systems, they could rely on these facilities to manage resources in such a way that resource consumers (that is, software components) can get a level of service that goes far beyond the traditional “best effort” policy.

JAMUS is an experimental platform that lies at the frontier between the two latter categories. It is dedicated to supporting the deployment of “untrusted” software components (such as application programs and applets), 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). These components 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 (the acronym JAMUS actually stands for *Java Accommodation of Mobile Untrusted Software*), as well as guaranteed QoS as far as resource availability is concerned.

Overview

Resource control in JAMUS is based on a contractual approach. Whenever a software component applies for being deployed on the platform, it 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. In return, whenever the platform accepts a candidate component, it promises to provide this 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 (see for example [4, 2, 5]) 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 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.

Implementation details

JAMUS implements a resource broker, whose role is to guarantee the availability of resources for hosted components. At startup, the broker is given a description of the available resources, as well as instructions on how these resources should be used by components. JAMUS provides a series of interfaces and classes that enable the specification of resource access conditions as “resource utilisation profiles” (see [9, 8] for details). Such profiles can thus be defined as Java code, and handled at runtime as standard Java objects.

Contract subscription is achieved as 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 this component must also be expressed as a set of resource utilisation profiles. These requirements are examined by the resource broker in order to decide if the component can be admitted on the platform. Admission control is based on a simple, yet effective resource reservation scheme: a component is admissible only if the resources it requires are available on the platform in sufficient quality and quantity. Moreover when a component is admitted the resources it required are “reserved” for its sole usage until it reaches completion.

When a candidate component has successfully passed the admission control test, it can start running on the platform. However, once a component has been accepted, it is still considered as non-trustworthy. 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 every component hosted by the JAMUS platform runs under the control of a dedicated component monitor, whose implementation relies on the monitoring and control facilities offered by RAJE (see [9, 8] for details). Whenever a monitor observes that the component it is in charge of is misbehaving, it can sanction the component. Sanctioning a component is done either by locking the resources considered, or by killing the component itself.

4 Discussion and perspectives

The development of both RAJE and JAMUS is still in progress. As a consequence, both products are 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 questions we’d like to answer in the future.

No extensive performance evaluation has been performed so far. However, preliminary experiments have shown that the overhead of resource monitoring in JAMUS remains quite low. For example, in order to evaluate the specific overhead of network activity monitoring, measurements have been realized with a Java-based FTP server[2] run as a component hosted (and constantly monitored) by JAMUS. During this

[2] JFTPd server developed by the MIT *(http://jftpd.prominic.org/index.html)*. Experiment achieved using two workstations (with PII processors) directly connected by a point-to-point 100 Mbps Fast Ethernet link.

experiment, file uploads and downloads realized by the server in these conditions showed no detectable degradation of performance.

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 makes 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.

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 *The 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[2] Nataraj Bagaratnan and Steven B. Byrne. Resource Access Control for an Internet UserAgent. In *The 3th USENIX Conference on Object-Oriented Technologie and Systems*, 1997.

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

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

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

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

[7] Li Gong and Roland Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Scurity*, March 1998.

[8] Nicolas Le Sommer and Frédéric Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD'2002, Berlin, Germany)*, June 2002. To be published.

[9] Nicolas Le Sommer and Frédéric Guidec. JAMUS: Java Accommodation of Mobile Untrusted Software. In *4th EurOpen/USENIX Conference (NordU'2002, Helsinki, Finland)*, February 2002. To be published.