Resource Consumption Interfaces for Java™ Application Programming − a Proposal

Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper

Sun Microsystems, Inc.
901 San Antonio Rd.
Palo Alto, CA 94303, USA
{Grzegorz.Czajkowski,Stephen.Hahn,Glenn.Skinner,Pete.Soper}@Sun.Com

Abstract

Software systems in many circumstances need an awareness of their resource impact on the underlying executing platform, so they can satisfy externally imposed performance requirements. Programs constituting such systems need the ability to control their consumption of resources provided by the platform. This document summarizes the current status of a proposal being prepared to define an extensible, flexible, and widely applicable resource consumption API for the Java platform. Control over resource consumption is developed using a set of abstractions that allow the statement of reservations and constraints for individual resources utilized by the executing application.

1 Introduction

The Java™ programming language [GJS+00] and its associated runtime environment have grown beyond their initial goal of writing portable applications. The advent of the Web, applications servers, and the enterprise and micro editions of the Java platform has created pressure to make more system programming features available to programmers, as they develop progressively more sophisticated applications in an increasingly wide range of environments. This document addresses one such need: the ability to monitor or control the consumption of resources by an application. The proposed resource consumption interface (RC API) controls resources available to collections of *isolates* and, as such, depends on the availability of the isolate abstraction. (The isolate abstraction is provided by Java Specification Request (JSR) 121 [JSR121]; a short overview of the Isolate API is given in the Appendix I). However, the RC API is designed so that new methods need not be added to the Isolate API.

The general goals for the resource consumption API are as follows:

- *Extensibility*. The resource consumption interface should support the addition of new controlled resources.
- *Cross−platform applicability*. The interface, as well as its underlying abstractions, should be applicable to all kinds of Java platforms, from the J2ME platform to the J2EE platform.
- *Cross−scenario applicability*. The interface should support the different forms of application management supported by the Isolate API, as well as being meaningful in a single application context.
- *Flexibility*. The interface should be able to describe and control a broad range of resource types.
- *Completeness of abstraction*. The interface should hide from applications whether a given resource is managed by the Java Virtual Machine (JVM™), by a core library, or by an underlying operating system (OS) facility.
- *Lack of specialization*. The interface should not require an implementation to depend on specialized support from an OS or from hardware, although implementations may take advantage of it if available.

The proposal examines the resource consumption interfaces from the perspectives of three classes of developers, each of whom is a participant in resource management: the application developer, the middleware developer, and the runtime environment developer. Each of these views is considered after an overview of the API. This is work in progress, and at the time of this writing, there are no plans for making it a part of the Java platform.

2 Interface overview

The interface description starts with resource classes and proceeds to the domain abstraction, which relates resources to controlled isolates and their usage constraints and reservations. The discussion then covers a constraint trigger callback mechanism and finishes with a description of the relationship between the interface and the security model. Refer to Appendix II, which summarizes the RC API, when reading the following sections.

2.1 Resources

A resource is a measurable entity that a program needs such that a shortfall results in a performance change. Implementations of the Java™ runtime environment (JRE) differ in their ability to expose resources through this API. An implementation makes the resources it does expose available as subclasses of the abstract Resource class. Each such subclass completely describes the general behavior of the resource and the particulars of its management (e.g. the precision of its measurement). Some or all of these would typically be available in the documentation of a particular JVM implementation, and the intent of the RC API is to expose them to programmers. Resources that are differently managed and accounted for should be described as different subclasses of Resource. For instance, the accumulated CPU time is a different resource from share (rate, bandwidth) of the CPU time. Instances of some subclasses of Resources can be parameterized; for example, a network bandwidth resource can be associated with a specific port. It is expected that the cases of parameterizable resource will be rare. Fully qualified names of subclasses of Resource serve as resource names, similar to the way security permissions are identified by their names.

The question of whether two resources are the same is answered by the equals() method. Each Resource subclass should define this method such that r1.equals(r2) is true if and only if both r1 and r2 refer to the same resource. For unparameterized resources, such as heap memory or absolute amount of CPU time, this this requirement is tantamount to a type equivalence test. For parameterized resources, this test should involve the equivalence of parameters; for instance, two instances of a parameterizable type NetworkBandwidth would be distinct if they related to different ports.

The usage of each resource should be expressible as a long integer, and integer comparison should be sufficient to tell whether two values are the same or one of them is greater than the other. The interface uses the Units Specification API [JSR108] to express the units in which resources are measured. In particular, the getUnit() method of Resource returns a description of the unit, which may be expressed in several different systems (e.g., metric, US, etc.) and which can contain standard scaling prefixes (e.g., *milli*, *kilo*, etc.).

2.2 Resource domains

Instances of Resource subclasses in themselves do not dictate the quantities of resources that an isolate can use. To this end, the interface introduces two other abstract classes: ResourceConstraint and ResourceReservation. Constraints specify threshold values of resource use [HTM01]. Exceeding a constraint triggers a customizable callback. Reservations specify that a resource is guaranteed in a given quantity. Each constraint governs the use of one resource only; similarly, each reservation reserves only one resource.

Multiple isolates can be bound to the same resource constraint, in which case their collective usage of a resource is constrained by the same value. Similarly, multiple isolates can be bound to the same resource reservation, in which case the isolates draw from the same reserved pool. Each isolate may be bound to multiple constraints and to multiple reservations; the only requirement is that it is not bound to more than one reservation for the same resource (there can be multiple constraints for the same resource).

Since ResourceConstraint and ResourceReservation have certain features in common, they are a subclass of ResourceDomain; thus from now on, "resource domain" or "domain" refers to either a constraint or a reservation. ResourceConstraint and ResourceReservation are abstract classes and should be extended in concert with defining the subclass of Resource on which they depend. One can create constraints and reservations via the newConstraint() and newReservation() methods of Resource.

The interface does not make it obligatory for any JRE implementation to manage any resources; thus it does not dictate any concrete subclasses of Resource. There is no requirement that a resource that can be constrained by the JRE be reservable, and vice versa. The interface provides a method for discovering which reservations or constraints can be imposed. Implementations may, but are not required to, support changing the values of constraints and reservations dynamically. The interface allows for querying whether or not this is possible.

Reservable resources can only be obtained if the requesting isolate is bound to an appropriate reservation. This is similar to the behavior of the Java 2 Platform security model: the lack of a defined permission means denial of access. The lack of binding to a constraint for a resource has the opposite semantics: no constraint means that the request should be granted, provided the resource is available. Binding an isolate to a reservation but not to a constraint for a resource means that at least the reserved quantity is guaranteed, but more can be used if available; binding an isolate only to a constraint only means that no more of the resource than the constraint value can be used, but no quantity is guaranteed.

Resource accounting is based on resource domains; it is possible to query how much of a given resource is used by all isolates bound to a given domain, but it is impossible to ask how much of the resource a specific isolate has consumed/is consuming unless it is the only one bound to the domain.

Resource domains allow for reserving and constraining resources available to collections of isolates; the aim is to provide a programmatic way to partition resources available to virtual machine(s) among Java applications. It is not the goal of this API to control the manner in which virtual machines obtain resources from the underlying OS (this might require defining additional primitives for the JVM−OS interaction). Similarly, this API does not control the manner in which virtual machines manage resources internally (this might require exposing the details of specific algorithms chosen by the JVM implementers, such as the garbage collector).

The first isolate is bound to all the reservations that the JRE implementation at hand manages. Implicitly, there are no constraints on the first isolate smaller than these initial reservations. All these reservations define maximal quantities of the resource available from the underlying platform. One way to create a new reservation is to specify that the value of the reservation be drawn (subtracted) from the current reservation for this resource. The reservations should add up to at most a maximum value dictated by the underlying platform. There is no such requirement on constraints. Initial reservations are themselves bound by the behavior built into the host operating environment with respect to resources. That is, implementations will allow reservations and constraints that operate within the resource limits imposed by the JRE and underlying operating environments that the JRE must use. Corresponding to the initial reservations, implementations may impose "sentinel constraints" for a given resource to express these kinds of limits.

2.3 Callbacks

The proposal defines a way to invoke an arbitrary action when a constraint triggers. Whenever a constraint is about to be exceeded, the callback will be invoked and subsequent requests for the resource block until the callback code returns. Depending on the return value of the callback, the offending request is granted or not. Callbacks can, for instance, lower resource consumption and then grant the request, terminate select isolates, throw an exception, or just deny the request.

In addition to hitting a definable constraint value, another kind of failure is possible: failing to provide a resource because there is just no more of it. This can happen when, for example, the JRE knows how to constrain the resource but does not provide for its reservations. Each of these kinds of failures is properly communicated to the appropriate isolate(s).

Isolates can also specify an arbitrary action to be performed when certain resource consumption situations arise. Such notification actions are always invoked asynchronously. Notifications can be thought of as constraints without direct consequence; hitting a constraint invokes an action (pre−defined or custom) which will determine whether the resource request will be granted or not, whereas a notification never stops a request from being granted.

Setting one or more constraints to be lower than a reservation allows for implementing low watermarks, traffic−light models of simple resource availability, and other simple "health monitors"; the trick is to set up the callback to warn the isolates bound to the constraint that the usage is high, giving them an opportunity to use less of a resource and avoid exceeding the reservation.

There is no requirement that all constraints and reservations to which a given isolate is bound be set by the same entity. Isolates can impose constraints on themselves so that they can react to triggered constraints. Privileged isolates can impose constraints on other isolates, and thereby act as as a resource manager for a set of isolates.

The details of dealing with callbacks and notifications depend on the outcome of JSR−121.

2.4 Security

Appropriate security permissions will guard the use of the API so that only privileged isolates can invoke sensitive methods of the interface. It is important to note that the API does not change in any way the existing Java 2 Platform security model, nor is the API associated with the standard security permissions. However, for some resources, an overlap in responsibilities may exist since a reservation may be viewed as a privilege to use the resource and a constraint as an explicit negative privilege (lack of privilege) to use the resource. Nevertheless, the correspondence is not one−to−one. For instance, a reservation for opening 50 files does not specify which files; a permission to open any file under /tmp does not mean that 10000 file descriptors are available for the task having this permission, even if that many files are located in directories located in /tmp. However, these cases of overlap are a result of the fact that trying to access a security−sensitive resource (e.g. file or socket) may still fail because of a lack of necessary resources. The RC API provides a controlled and uniform way of dealing with such situations. The RC API does not mandate that each resource has a corresponding security permission. For example, it seems counterproductive to require security permissions for CPU time and heap memory, but it is quite useful to apply resource controls to these resources.

3 The application programmer view

Existing applications can execute without any modifications, regardless of whether or not a JRE provides the RC API. Such executions are "passive" with respect to the API; the application does not take any advantage of the new capabilities. New applications, written against the API, may inquire about the resource reservations available to them, about the constraints imposed on them, and about their resource consumption related to the resource domains to which they are bound. Such "proactive" applications may then react to the information, possibly improving their performance, availability, and overall resource utilization.

For instance, an application may require notification whenever its heap memory usage exceeds a certain threshold, and upon receiving the notification, it may remove some items from its private in−memory cache in order to lower its consumption of heap memory and thus avoid violating a constraint. Another application may be interested to know how much network bandwidth there is reserved for it, so that it can decide whether it should transmit data out in an uncompressed format .

The following code demonstrates how a program can ask about all of its constraints and then register a notification callback to be invoked whenever the usage of each constrained resource reaches 90%:

```
Isolate me = Isolate.getCurrentIsolate();
ResourceConstraint[] cs = ResourceConstraint.getConstraints(me);

NotificationCallback callback = new NotificationCallback() {
  public void run(Resource r, long usage, long value, boolean up) {
    System.out.println("Almost no " + r + " left!"); // do something about it!
  }
};

for (int i = 0; i < cs.length; i++) {
  ResourceConstraint c = cs[i];
  Resource r = c.getResource();
  long v = c.getValue();
  System.out.println("Constraint on " + r + ":" + v + " " + r.getUnit());
  // 2nd arg: when the consumption is increasing; 3rd arg: each time this happens
  c.registerNotificationInterest((long)(0.9 * v), true, false, callback);
}
```

4 The middleware developer view

Certain features of the RC API are useful for "middleware," such as application servers, servlet−enabled Web servers, and various other applications that manage the execution of other isolates; these environments will be referred to as "application managers." For an application manager, the RC API is a mechanism by which resource consumption policies can be put in effect. In particular, application managers (and other privileged applications) can take advantage of the RC API to create new resource domains, destroy resource domains, bind isolates to resource domains, re−bind isolates to resource domains, and obtain the creating isolate of a given resource domain. The bindings between isolates and domains can be arranged such that co−operating computations (isolates) share the same pool of resources by virtue of being bound to the same set of reservations. Constraints may be imposed on isolates to prevent them from monopolizing the use of a certain resource, thereby preventing some kinds of denial−of−service attacks. An application manager may monitor resource consumption and use this information to decide whether a new request to start an application should be accepted or rejected. Possibilities are virtually endless.

The following example illustrates how middleware can use the proposed interface to create two isolates which will be bound to the same reservation and the same constraint for heap memory, both for 1024 KB. However, they will each be bound to a different constraint for CPU time (10s and 100s, respectively). The callback will terminate all isolates bound to a constraint when invoked. The code to check whether required constraints and reservations are implemented and any needed conversion of values (e.g. from seconds to milliseconds) is not shown. For resources for which neither a constraint nor a reservation was expressed, such as network resources, the use is unconstrained, but the availability is not guaranteed. The resulting organization of isolates and their bindings to resource domains is shown in Figure 1.

```
ConstraintCallback callback = new ConstraintCallback() {
  public boolean run(ResourceConstraint constraint, long usage) {
    Isolate[] is = constraint.getIsolates();
    for (int i = 0; i < is.length; i++) is[i].doHalt(AppMgr.NORESOURCE);
    return false;
  }
};
```

```
Isolate isolate1 = new Isolate(mainClassName1, arguments1, ...);
Isolate isolate2 = new Isolate(mainClassName2, arguments2, ...);

Resource heap = Resource.newInstance("somepackage1.HeapMemory");
Resource cpu = Resource.newInstance("somepackage2.AbsoluteCpuTime");
ResourceReservation heapR = heap.newReservation(1024);
ResourceConstraint heapC = heap.newConstraint(1024, callback);
ResourceConstraint cpuC1 = cpu.newConstraint(10, callback);
ResourceConstraint cpuC2 = cpu.newConstraint(100, callback);

ResourceDomain.bind(isolate1, heapR);
ResourceDomain.bind(isolate2, heapR);
ResourceDomain.bind(isolate1, heapC);
ResourceDomain.bind(isolate2, heapC);
ResourceDomain.bind(isolate1, cpuC1);
ResourceDomain.bind(isolate2, cpuC2);

isolate1.start();
isolate2.start();
```

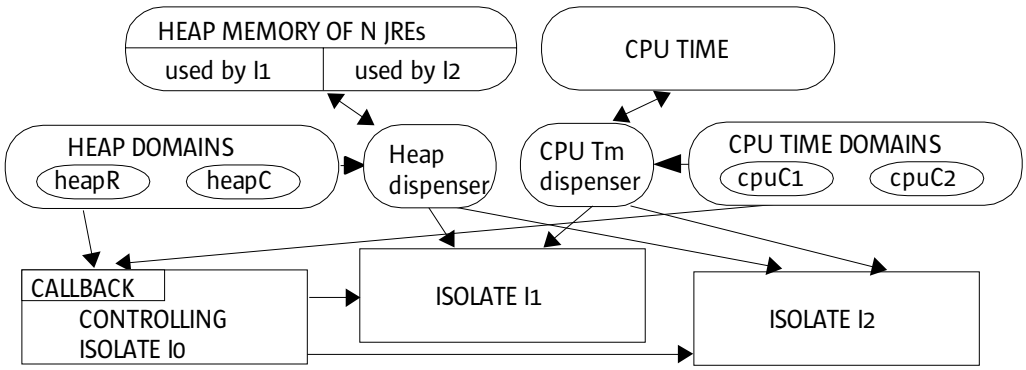

Figure 1. Middleware usage example.

5 The runtime environment developer view

The two sections above discussed how non−privileged and privileged applications can make a use of the API. The task of providing an actual implementation within this framework belongs to virtual machine implementers and to developers of libraries defining resources.

The single resource R (e.g. heap memory) will be the focus of this description. If D is a "dispenser," which is an entity either in the JRE or in a library that "knows" how to grant requests for a resource (e.g. an automatic memory management subsystem in the JRE's runtime), the following steps have to be performed by an implementer of the interface for this particular resource R:

- Implement a concrete subclass of Resource to represent R.
- Implement D so that it can properly grant access to the resource, understanding the constraints and reservations imposed by the use of the RC API by the privileged applications.
- Apply the appropriate optimizations.

For example, to make sure that code from Sections 4 and 5 can execute, the classes somepackage1.HeapMemory and somepackage2.AbsoluteCpuTime extending Resource must be defined. The required constraints (on heap memory and on absolute CPU time) must be defined too, as well as resource reservation on heap memory.

These new classes should interact with the resource dispenser so that constraint and notification callbacks are properly triggered and so that creating a reservation guarantees a required resource amount. For instance, a list of callbacks sorted by their "CPU time used to trigger" value may be maintained by classes related to the absolute CPU time. The periodic check on how much CPU time has actually been used would then result in triggering the callbacks whose value has already passed.

Finally, optimizations should be applied. for example, implementing constraints on heap memory should not negatively impact high−performance garbage collectors, and a simplistic approach of checking against

constraint(s) on each object allocation should be replaced by exploiting the features of modern garbage collectors, such as a generational heap layout, to limit the frequency of constraint checking to garbage collection times.

6 Conclusions

The proposal outlined in this paper defines an application programming interface for controlling the consumption of resources for the Java platform. The features distinguishing this proposal from prior related work [CvE98,BHV01] are the API's extensibility, flexibility, and wide applicability. We are currently working on substantiating these claims with concrete implementation.

Acknowledgments. The authors acknowledge valuable input from Laurent Daynes, Bill Foote, Hideya Kawahara, Tim Lindholm, and Gary Pennington of Sun Microsystems, as well as the JSR 121 Expert Group.

7 References

[BHL00] Back, G., Hsieh, W., and Lepreau, J. *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.* In Proceedings of 4th OSDI, San Diego, CA, October 2000.

[BHV01] Binder, W., Hulaas, J., and Villazon, A., *Portable Resource Control in Java: The J−SEAL2 Approach.* ACM OOPSLA'01, Tampa, FL, October 2001.

[CvE98] Czajkowski, G., and von Eicken, T. *JRes: A Resource Control Interface for Java.* In Proceedings of ACM OOPSLA'98, Vancouver, BC, Canada, October 1998.

[CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution*. ACM OOPSLA'01, Tampa, FL, October 2001.

[DBC+00] Dillenberger, W., Bordwekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., and St. John, R. *Building a Java virtual machine for server applications: The JVM on OS/390.* IBM Systems Journal, Vol. 39, No 1, 2000.

[GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification.* 2nd Ed. Addison−Wesley, 2000.

[HTM01]] Hahn, S., Tucker, A., Marsland, T. *Resource management mechanisms in Solaris.* Sun Users Performance Group, Amsterdam, 2001.

[JSR108] http://jcp.org/jsr/detail/108.jsp.

[JSR121] http://jcp.org/jsr/detail/121.jsp.

8 Appendix I: What is an Isolate?

The interface proposed here uses the isolate as its unit of control with respect to resource consumption. An isolate encompasses and separates the objects and mutable static state of an application or application component. Isolate creation and life cycle management are provided by the Application Isolation API ("Isolate API"), the formal output of JSR−121.

The Isolate API holds the view of an application from a management standpoint constant across a wide range of JRE implementations. By virtue of different API implementations, an application may reside by itself within a conventional JRE while possessing additional isolation by virtue of operating system process boundaries. It may reside within a process−based JRE whose infrastructure is largely shared with other JRE instances, each in their own process. It may reside with other isolated applications within a single JRE instance. In all cases, the VM as an abstraction remains constant from the standpoint of Java state while the JRE implementation varies. The intent is to cover the whole spectrum of sharing among JREs, from no sharing through intermediate sharing models to more aggressive approaches to sharing [DBC+00,BHL00,CD01].

In addition to simple life cycle management the Isolate API will provide a simple inter−isolate communication mechanism, and support passing the initial set of properties, preferences (see JSR−10) and other context required by a new application created within an isolate.

The Isolate API will be fully compatible with existing applications and middleware. In particular, applications written before JSR−121 may be managed by the API without the need for modification, and likewise, middleware can be unaware of or ignore the isolate API.

9 Appendix II:

The listing below summarizes the proposed API. For brevity, exceptions and comments are omitted.

```
public abstract class Resource {
  protected Resource();
```

```
  protected Resource(String[] parameters);
  public static final Resource newInstance(String name, String[] parameters);
  public static final Resource newInstance(String name);

  public abstract String getName();
  public String[] getParameters();
  public abstract ucar.units.Unit getUnit(); // JSR-108 type
  public abstract long getMeasurementPrecision();
  public abstract long getMeasurementDelay();

  public abstract ResourceConstraint newConstraint(long val, ConstraintCallback c);
  public abstract ResourceReservation newReservation(long val);
}

public abstract class ResourceDomain {
  protected ResourceDomain(long val);
  protected static final ResourceDomain[] getDomains(Isolate isolate);

  public static final void terminate(ResourceDomain domain);
  protected abstract void doTerminate();
  public final boolean isTerminated();

  public Isolate[] getIsolates();

  public abstract Object registerNotificationInterest(long resourceUsage,
    boolean up, boolean oneTimeOnly, NotificationCallback callback);
  public abstract void unregisterNotificationInterest(Object token);

  public static final void bind(Isolate isolate, ResourceDomain domain);
  protected abstract void doBind(Isolate isolate);
  public static final void unbind(Isolate i, ResourceDomain d);
  protected abstract void doUnbind(Isolate i);
  public static final void rebind(Isolate i, ResourceDomain fr, ResourceDomain to);
  protected abstract void doRebind(Isolate isolate, ResourceDomain to);

  public abstract long getPrecision();
  public long getUsage();
  public abstract Resource getResource();

  public abstract boolean canIncrease();
  public abstract boolean canDecrease();
  public final long getValue();
  public final void setValue(long val);

}

public abstract class ResourceConstraint extends ResourceDomain {
  protected ResourceConstraint(long val, ConstraintCallback callback);
  public static final ResourceConstraint[] getConstraints(Isolate i);
  public ConstraintCallback getCallback();
  public abstract boolean enforcedBeforeViolation();
  public static final void registerResource(String className);
  public static final String[] getConstrainable();
}

public abstract class ResourceReservation extends ResourceDomain {
    protected ResourceReservation(long val) {
    public abstract ResourceReservation spawnNewReservation(long val);
    public static ResourceReservation[] getReservations(Isolate i);
    public abstract void transferTo(ResourceReservation res, long val); //atomic
    public static final void registerResource(String className);
    public static final String[] getReservable();
}

public interface NotificationCallback {
  void run(Resource resource, long u, long val, boolean up);
}

public interface ConstraintCallback {
  boolean run(ResourceConstraint constraint, long val);
}
```