

# **A Resource Management Interface for the Java™ Platform**

Grzegorz Czajkowski  
Stephen Hahn  
Glenn Skinner  
Pete Soper  
Ciarán Bryce

# A Resource Management Interface for the Java™ Platform

Grzegorz Czajkowski, Stephen Hahn,  
Glenn Skinner, Pete Soper, and Ciarán Bryce

SMLI TR-2003-124 May 2003

## **Abstract:**

Software systems in many circumstances need awareness of their resource usage. Meeting performance requirements often requires the ability to manage consumption of resources provided by the environment. Resource management is traditionally handled by operating systems, but the growing need to use safe languages in the systems programming domain adds increased pressure to equip them with resource management capabilities at a level of abstraction that fits gracefully with the language.

This paper proposes an extensible, flexible, and widely applicable resource management interface for the Java™ platform. The interface is small, but capable of modeling a variety of resources and resource management policies. In particular, application-specific resources can be defined, as well as more traditional computational resources such as heap memory and processor time. The interface is presented here in detail, along with a series of examples and a description of a prototype implementation.



M/S MTV29-01  
2600 Casey Avenue  
Mountain View, CA 94043

## **email addresses:**

grzegorz.czajkowski@sun.com  
stephen.hahn@sun.com  
glenn.skinner@sun.com  
pete.soper@sun.com  
bryce@cui.unige.ch

© 2003 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, Java HotSpot, JDK, J2EE, J2ME, JVM, JSR 200, Solaris 9 Operating System, and Sun Enterprise 3500 are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <[jeanie.treichel@eng.sun.com](mailto:jeanie.treichel@eng.sun.com)>. All technical reports are available online on our Website, <http://research.sun.com/techrep/>.

# A Resource Management Interface for the Java™ Platform

Grzegorz Czajkowski, Stephen Hahn,  
Glenn Skinner, Pete Soper

Sun Microsystems Laboratories  
2600 Casey Avenue  
Mountain View, CA 94043, USA  
{firstname.lastname}@sun.com

Ciarán Bryce

Object Systems Group  
University of Geneva  
Switzerland  
bryce@cui.unige.ch

## 1 INTRODUCTION

The Java programming language [GJS+00] and its runtime environment have grown beyond their initial goal of writing portable applications. The advent of the Internet, application servers, and the enterprise and micro editions of the Java platform have created pressure to make more system programming features available to programmers, as they progressively develop more sophisticated applications for an increasingly wide range of environments.

An area where safe language platforms seriously lag behind operating systems is that of resource management (RM) facilities. Preventing denial of service attacks, providing load balancing, and monitoring the usage of a given resource are all difficult to do at all and impossible to do well currently in the Java platform. In general, these and other features cannot be provided without going beyond the safe language through mechanisms such as native code or shell scripts that ask the OS to handle RM-related matters. The lack of a standard, programmatic way to partition resources available to virtual machine(s) among Java applications has led to a number of awkward, ad-hoc solutions, limiting the expressiveness of safe languages and discouraging some developers from using them.

This document proposes a resource management interface (RM API) for the Java platform. It is set apart from the previous efforts in this area [CvE98,BHV01] by the following features:

- *Wide applicability.* The interface is applicable to a variety of resource management scenarios and allows the expression of numerous resource management policies.
- *Flexibility.* The interface enables managing a broad range of resource types.
- *Extensibility.* The resource management interface supports the addition of new resources in a uniform manner.
- *Completeness of abstraction.* The interface hides from applications whether a given resource is managed by the underlying operating system (OS), by the Java Virtual Machine (JVM™), by a core library, or by trusted middleware code. As a consequence, resource policies can be written abstractly for resources exhibiting common behavior, without regard for how those resources are implemented.

- *Lack of specialization.* The interface does not require an implementation to depend on specialized support from an OS or from hardware although implementations may take advantage of such support if available.
- *Cross-platform applicability.* The interface, as well as its underlying abstractions, is applicable to all kinds of Java application environments, from the J2ME™ platform to the J2EE™ platform.

A *resource* is a measurable entity that a program needs such that a shortfall results in a performance change; examples include heap memory, the number of database connections or server threads in use, and processor time. A resource management policy for a resource defines when a computation may gain access to, or *consume*, a unit of that resource. This proposal allows an expressive set of resource management policies to be coded. A computation can be dynamically bound to such a policy. Programs can reserve resources in advance and thus ensure predictable execution. Applications can install resource monitoring code so that proactive programs can observe resource availability and take any actions required to ensure performance and availability or to ward off denial of service attacks. Existing applications can still run without modification, even if the Java Development Kit (JDK™) classes or Java runtime environment (JRE) they depend on exploits the resource management framework. The interface does not impact in any way how actual resource managers (schedulers, automatic memory subsystems, etc.) should be written – these managers become RM-enabled by inserting calls to RM API methods to convey information about resource consumption attempts. The effort required to retrofit existing resource managers is thus minimized.

The unit of management for the RM API is an *isolate*, the abstraction recently introduced by Java Specification Request (JSR) 121 [JCP01b] (see Appendix I for an overview). An isolate is an encapsulated Java program or application component that shares no state with other isolates. Isolates have proven to be a convenient abstraction on which to develop resource management: isolation allows unambiguous resource usage accounting and clean reclamation upon program termination. The Isolate API can be delivered using a range of techniques, from equating an isolate with an OS process executing the JVM, through JVM implementations with transparent cross-process meta-data sharing, to multitasking virtual machines. The RM API maintains this property: it is applicable to all these

implementation choices, and in no part does it depend on the underlying architecture of the virtual machine.

The remainder of this paper is organized as follows. Section 2 introduces the key concepts of the API. Section 3 covers details. Section 4 analyzes what happens during a resource consumption request. Section 5 provides examples of how resources are exposed through the RM API. Section 6 describes a prototype implementation. Section 7 discusses design decisions and Section 8 discusses related work. Finally, Section 9 summarizes the paper.

## 2 MAIN ABSTRACTIONS

The main abstractions of the resource management interface are introduced below. Details follow in the next section.

With the RM API, each resource is described by a set of *resource attributes*. These define properties of the resource and its implementation – for example, whether resource availability is unlimited and whether reservations can be set on the resource. The fully qualified name of a resource's attributes class serves to identify the resource unambiguously.

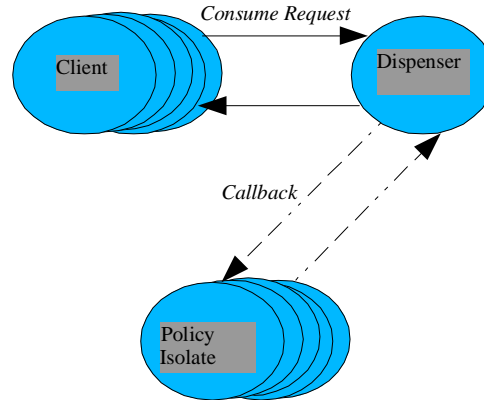
The bridge between the resource management interface and the code that actually implements (fabricates) a resource is a resource *dispenser*. The dispenser monitors the amount of the resource available to computations.

A resource consumption policy is represented by a *resource domain*. All isolates bound to a given resource domain are uniformly subject to that domain's policy for the underlying resource. The policy may specify *reservations* (guaranteed resource availability) and *actions* that should execute upon specific usage events.

Certain kinds of actions influence the decision to grant a particular request to consume a resource; others notify interested parties of an occurrence of a specific event. To this end, actions contain *callbacks*, which can be written to be invoked immediately when a resource consumption request is made, or in response to a resource consumption decision, just before control returns to the requester. To provide control over when its callbacks are invoked, each action contains a *trigger*, which examines each resource consumption request to determine whether it merits intervention from the action's callbacks.

Figure 1 maps these concepts onto the isolates that interact with a given resource domain. One or more client isolates make requests to consume or release units of the domain's resource. The dispenser's isolate receives these requests and evaluates them against the resource domain's policy, which may require invoking callbacks that are part of actions established by zero or more managing isolates; the collective contributions of these isolates interact with the dispenser to determine the domain's overall policy. Although the figure shows the isolates in each role as being disjoint, it is possible for a single isolate to act in multiple roles.

The following example illustrates some of the notions described above. Commentary follows immediately after the code, and Figure 2 shows the resulting configuration of isolates and their binding to resource domains.

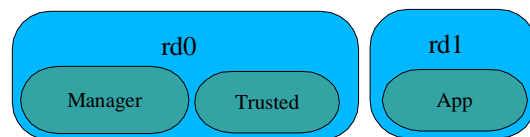


**Figure 1.** Client and policy-imposing isolates bound to a single domain.

```
public static void main(String[] args) { // class Manager
    String R = args[0]; // get name of resource to manage
    ResourceDomain rd0 = ResourceDomain.currentDomain(R);
    ResourceDomain rd1 = ResourceDomain.newDomain(R);
    long reservation = rd0.getReservation().getValue();
    rd0.setReservation(new Reservation(reservation - 100));
    rd1.setReservation(new Reservation(100));

    ConsumeCallback.Pre preCallback =
        new ConsumeCallback.Pre() {
        public long preConsume(ResourceDomain rd,
            long current, long proposed) {
            String name = rd.getResourceAttributes().getName();
            log("Reject " + (proposed - current) + " of " + name);
            return current; // veto the request
        }
    };
    Trigger trigger = Triggers.newAbsoluteUp(100);
    ConsumeAction action =
        new ConsumeAction(false, true, preCallback, trigger);
    rd1.setConsumeAction(action);
    Isolate iA = new Isolate("Trusted", new String[0]);
    rd0.bind(iA);
    Isolate iB = new Isolate("App", new String[0]);
    rd1.bind(iB);
    iA.start(new Link[0]);
    iB.start(new Link[0]);
}
```

Manager is the initial isolate, which is assumed to be bound to a domain for a resource denoted by a string variable R (see Sec. 6 for bootstrapping details). R can be any resource the particular implementation of the RM API delivers – the code to manage it is the same. This is an important feature of the interface: policies can be expressed abstractly, parameterized by resource names.



**Figure 2.** Example bindings of isolates to domains.

The manager obtains a handle `rd0` to its domain and creates another domain `rd1` for the same resource. Then it sets a reservation for 100 units of `R` on the new domain after lowering its own reservation by the same amount and sets a consume action. The action consists of a non-persistent (removed after the first execution; the `false` argument), synchronous (blocking the consume request; the `true` argument) pre-callback (“pre” indicates that it is invoked before requests to consume resources are granted) and a *trigger*, which determines under what circumstances the callback should be executed. We chose a predefined trigger that causes the action's callback to be executed when usage increases to or beyond the specified threshold value. The callback itself has three arguments: the domain against which the requested usage will be charged if granted (it is always the same domain on which the consume action has been set), the current usage, and the proposed usage. Returning the current usage value indicates that the request for an additional quantity (proposed minus current) is refused. This consume action constitutes a *constraint* that prevents isolates bound to `rd1` from using more than 100 units of `R`.

After completing this setup, the manager creates a new isolate `iA`, which will execute the Trusted main class and binds it to `rd0` – from now on the manager and `iA` will share `rd0`. This means that any usage of `R` by the manager or by Trusted is accounted against `rd0`, and the two isolates share the same resource management policy (reservations, consume actions, etc.) Finally, a new isolate `iB` executing main class `App` is created and bound to `rd1` (Figure 2); `iB` is thus subject to the policy the manager defined.

To see how isolates can take advantage of the RM API, consider the following `App` class:

```
public static void main(String[] args) { // class App
    String R = args[0]; // get name of resource to manage
    ResourceDomain rd = ResourceDomain.currentDomain(R);
    long reserved = domain.getReservation().getValue();
    if (rd.getIsolates().length != 1 || reserved < 50)
        error("I don't like this ...");

    ConsumeCallback cRed = new ConsumeCallback.Post() {
        void postConsume(ResourceDomain rd,
            long previous, long granted) {
            // Arrange to decrease consumption immediately!
        }
    };
    Trigger tRed = Triggers.newAbsoluteUp(reserved - 5);
    ConsumeAction red =
        new ConsumeAction(true, false, cRed, tRed);
    rd.setConsumeAction(red);

    ConsumeCallback cGreen = new ConsumeCallback.Post() {
        void postConsume(ResourceDomain rd,
            long previous, long granted) {
            // R abundant, OK to increase its consumption
        }
    };
    Trigger tGreen = Triggers.newAbsoluteDown(5);
    ConsumeAction green =
        new ConsumeAction(true, false, cGreen, tGreen);
    rd.setConsumeAction(green);
    // go about consuming ...
}
}
```

After obtaining a handle to its domain for `R`, `App` makes sure that there are no other isolates bound to it and that at least

50 units of the resource are available. It then creates two consume actions. Both are persistent, asynchronous, and “post,” which means that when they trigger, they are executed asynchronously immediately after the implementation commits to allowing (or denying) a resource consumption request. The red consume action triggers when usage is just five units below the reserved quantity; the goal of its associated callback is to inform the rest of the program that lowering its consumption of `R` is imperative. The green consume action has the dual goal: whenever `R` is abundant (its consumption drops to no more than five units), the imperative conservation state is rescinded, and the program may resume consuming `R` freely. Both of these actions behave as *notifications*; they inform the application of a change in its resource consumption state.

The examples above show ways to exert control over the amount of resource consumption, but they say nothing about controlling the rate of resource consumption. The reason for this omission is that the RM API provides no direct means of manipulating consumption rates; to do so would require extending the thread scheduler with interfaces for influencing its scheduling decisions and would require adding a set of rate-controlling methods to our API. Rather than following this course, we chose a different alternative, based on the observation that the ability to gain control at every resource consumption point implies the ability to delay the consuming thread at each of those points. Thus, to impose a desired consumption rate for a given resource, it suffices to throttle consumption requests until they match that rate. Section 3.4 contains an example that illustrates this approach.

### 3 DETAILS OF THE API

This section presents the details of the interface. The aim is to show that the API is expressive enough to capture a wide variety of application and system-level resources and policies.

#### 3.1 Resource Attributes

The API characterizes a resource as a set of attributes, grouped together in an instance of the `ResourceAttributes` class:

```
public abstract class ResourceAttributes {
    public abstract long getGranularity();
    public abstract long getMeasurementDelayMillis();
    public final String getName() { return getClass().getName(); }
    public abstract Unit getUnit();

    public abstract boolean isDisposable();
    public abstract boolean isReservable();
    public abstract boolean isRevokable();
    public abstract boolean isUnbounded();
    public static ResourceAttributes getInstance(String name);
    public static ResourceAttributes[] getRegistered();
}
```

To define a resource through the RM API, a programmer simply declares a subclass for the new resource and implements the abstract `is*()` and `get*()` methods. Each subclass describes the resource's behavior in a particular RM implementation.

The attributes are carefully chosen to provide programmatic means of learning about the properties of a given resource and to enable optimizations and early error detection in RM API implementations. The only final method, `getName()`, is defined to be the name of the specific subclass of `ResourceAttributes`. To guarantee predictability of execution, programmers defining a new resource should give the corresponding `ResourceAttributes` subclass a unique package name and ensure its instances are immutable.

The four boolean attributes determine the semantics of handling consume requests. A resource is *disposable* if it is possible to identify a span of program execution over which a given resource instance is considered to be consumed. Outside of this span, the resource instance is available for (re)use. As a consequence, usage is not necessarily monotonic. A page of memory is a disposable resource; CPU time is not. An example of the usefulness of this attribute is in allowing *unconsuming* (i.e., returning to the pool of resources) of disposable resources only. The same operation for a non-disposable resource is erroneous.

A resource is *revokable* if units of the resource previously granted to the resource consumer can be withdrawn without affecting the consumer's behavior, except possibly for its rate of progress. An example is physical memory: the OS can alter the size of the page frame pool it dedicates to a process's address space without the process noticing.

A resource is *unbounded* if there is no fixed limit on the amount available. For example, in the absence of a constraint (perhaps issued by the underlying host platform), "absolute CPU time" is an unbounded resource.

After a successful reservation request of a *reservable* resource, it is guaranteed that the system is able to supply the reserved units of resource. This does not imply that a client may consume the resource, as that is also dependent on the resource usage policy of the client's resource domain. The definition is phrased in terms of resulting usage, rather than in terms of number of units requested. This distinction is important for disposable resources, since the sum of requested units might overstate actual usage. Any resource can be reservable although attempting to reserve an unbounded resource seems rather pointless (the API does not prevent it, though).

The interface requires that quantities of resources (e.g., usage, reservations) be expressible as long integers. Integer comparison must 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 [JCP01a] to define the units in which resources are measured. In particular, the `getUnit()` method of `ResourceAttributes` 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.).

The *granularity* of a resource is the indivisible amount of the resource in a given implementation. For instance, a heap might be managed as a set of pages; in this case, although the resource's unit is bytes or kilobytes, the deliverable granularity is the underlying system's page size, e.g., four kilobytes. RM API method arguments that specify resource quantities are automatically rounded up to the nearest multiple of granularity. Thus, requesting to consume a non-

multiple of granularity is legal, but the returned value will be a granularity multiple.

The *measurement delay* is the maximum amount of time that can pass between resource consumption and updating the usage information. For example, controlling the number of open file descriptors can be done accurately at any time (measurement delay is zero), whereas controlling CPU time usage via sampling once a second has a measurement delay of one second. An important implication of measurement delay is the possibility of uncontrolled consumption during the delay interval. If this is undesirable, resource providers should make measurement delay of the resource in question as small as possible.

The rationale behind granularity and measurement delay is to allow managing the tradeoff between accounting precision and its cost. Both attributes are important for resource providers; applications are typically insulated from directly dealing with them (Sec. 5).

While designing the interface, we experimented with many more attributes, but only the ones presented above passed the test of requiring manifestation in the API. An example of a property that seemed interesting is being *explicit*: a resource is explicit if it is possible to identify a proper subset of the resource consumer's bytecodes such that a bytecode in the subset corresponds to a point at which the resource is consumed. A file descriptor is an example of an explicit resource; CPU time is not explicit. This property is important in determining whether and where in the program error handling related to resource shortage should be placed. However, this is simply a matter for a resource's documentation; it has no impact on the RM API itself.

### 3.2 Resource Domain

A *resource domain* encapsulates a usage policy for a resource. All isolates bound to a specific resource domain are subject to the same usage policy. The RM API does not impose any policy on a domain; policies are explicitly defined by programs.

An isolate cannot be bound to more than one domain for the same resource. Nevertheless, an isolate can be bound to many resource domains, where each domain controls a *different* resource. Thus, two isolates can share a single resource domain for, say, heap memory, but be bound to distinct domains for outgoing socket traffic.

The public methods of the class are as follows:

```
public final class ResourceDomain {
    public static ResourceDomain[] currentDomains();
    public static ResourceDomain currentDomain(String name);
    public static ResourceDomain newDomain(String name);
    public ResourceAttributes getResourceAttributes();
    public void bind(Isolate isolate);
    public void unbind(Isolate isolate);
    public Isolate[] getIsolates();
    public long consume(long quantity);
    public long consumeAllOrNothing(long quantity);
    public long unconsume(long quantity);

    public void setConsumeAction(ConsumeAction action);
    public void removeConsumeAction(ConsumeAction action);
}
```

```

public Reservation getReservation();
public void setReservation(Reservation reservation);

public long getUsage();
public long getTotalUsage();
public long getTotalQuantity();
public long getTotalReservedQuantity();

public void terminate();
public boolean isTerminated();
}

```

The static methods of the class return the set of domains to which the current isolate is bound, return a specific current domain given the resource name (throwing an exception if the resource is registered but not bound in the current isolate – Sec. 3.3), and create a new domain. The attributes for the resource for which a domain is created are obtained via `getResourceAttributes()`. This example shows how an isolate can discover all the domains it is bound to:

```

ResourceDomain[] rds = ResourceDomain.currentDomains();
for (int i = 0; i < rds.length; i++) {
    String name = rds[i].getResourceAttributes().getName();
    System.out.println("I am bound to " + name);
}

```

The `bind()` method binds an isolate to a resource domain. This method fails if the isolate is already bound to a domain for the same resource. The `unbind()` method only succeeds when the isolate has been terminated or when its consumption of the resource is zero. An array of isolates bound to a given domain can be obtained via `getIsolates()`. This is useful, for example, in determining whether an isolate is the only one bound to the domain and, consequently, the only one subject to the given resource management policy.

Any isolate bound to a resource domain can request to *consume* units of the resource as well as *unconsume* units previously obtained, provided that the resource is disposable. These operations typically invoke core or middleware code implementing the resource. For example, client applications opening and closing sockets remain unchanged, but if the `java.net` code is extended with RM facilities, some of the socket operations may invoke `consume()` and `unconsume()` on the client's resource domain.

The `consume()` method can return less of a resource quantity than requested, subject to the decision made by the dispenser (Sec. 3.3), which, in turn, is made based on consume actions (Sec. 3.4) and reservations (Sec. 3.5). Such partial grants of requests may be acceptable for certain resources. If an entire requested quantity is necessary for a given operation to succeed, resource implementations should use `consumeAllOrNothing()` (e.g., an attempt to allocate a 1MB array should fail if only 512KB of heap memory can be allocated). Invoking this method does not guarantee the success of the request, but does prevent futile partial request satisfaction.

Resource management policies are dynamically set by setting and removing consume actions and reservations on resource domains. Removing a reservation is done by setting the reserved value to 0. Applications can learn about the quantities reserved.

The `getUsage()` method returns the number of resource units consumed by the domain. Three methods are provided to obtain information related to all domains associated with the same dispenser as the target domain: `getTotalUsage()` returns the total amount of consumption, `getTotalQuantity()` returns the total amount of the resource in care of the dispenser, and `getTotalReservedQuantity()` returns the sum of all reservations on the domains associated with the dispenser. These methods are particularly useful in determining how large new reservations can be.

All usage and reservation statements are with respect to resource domains, and the interface makes no provisions for distinguishing consumption and reservations within isolates bound to the same resource domain. Thus, it is not possible to know how much a given isolate is consuming of a given resource unless it is the only isolate bound to its resource domain (Sec 7.1).

### 3.3 Dispenser

A *dispenser* controls the quantity of a resource available to resource domains and, thus, (indirectly) to computations. It is important to stress that dispensers do not themselves implement resources. A resource's implementation consults a dispenser upon a request for a resource via the `consume()` method of `ResourceDomain`. The dispenser indicates how much of a resource can be granted based on the current usage and policy. Multiple resource domains can be associated with a dispenser. Such domains may have different policies, and the sets of isolates bound to each are disjoint but are collectively subject to certain invariants that the dispenser enforces. For example, the sum of all reservations across the domains cannot be greater than the amount of the resource the dispenser controls. The interplay between resource managers, dispensers, and domains is explained in Section 4. The `Dispenser` class is as follows:

```

public abstract class Dispenser {
    public static Dispenser newInstance(String name,
                                       String[] args);
    public static void registerDispenser(Dispenser dispenser);
    public static void unregisterDispenser(Dispenser dispenser);
    protected boolean isGlobal();
    protected final void setTotalQuantity(long totalQuantity);
    protected ResourceAttributes getResourceAttributes();
}

```

Dispensers are created by invoking `newInstance()`. This method takes as arguments the name of a `Dispenser` subclass, along with dispenser-specific arguments that should be described in the documentation for concrete dispensers. If the managed resource is not unbounded (e.g, this is typically the case for heap memory), the constructor should inform the dispenser about the total quantity available via `setTotalQuantity()`. Internal `Dispenser` code uses this value to determine whether a reservation can be set. Once the dispenser is created, it can be registered – from that moment until de-registration, the dispenser is *active*, and resource domains can be created that feed off the dispenser.

There are two kinds of dispensers. A *global* dispenser is shared by all isolates making use of the resource it controls.

There is only one global dispenser per resource per system. However, there can be multiple instances of *local* dispensers in the system, but not more than one for the same resource in any one isolate.

The rationale for global dispensers is to model resources with a single source of “production.” Some examples are heap memory in a single-heap system and the number of open sockets. Local dispensers are created for resources that may be independently manufactured by multiple sources. An example is execution of several web servers in a single virtual machine where each instance of the web server and each servlet [Hall00] is a separate isolate. Each server may independently control the maximum number of concurrent requests each of its servlets can execute. This is a resource easily expressed in the proposed API, and its dispensers ought to be local to each server, because the servers need not coordinate with each other.

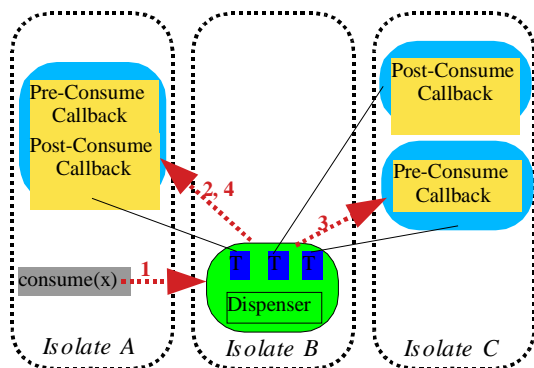
Creating a resource domain includes associating the domain with its dispenser. For global dispensers, this simply means looking up *the* dispenser. For local dispensers, the association procedure checks, in turn, whether there is a dispenser in the current isolate, in its parent, in the parent’s parent, and so on. The domain becomes associated with the first found dispenser. Creation fails if the search does not find a dispenser.

Defining a new dispenser requires sub-classing the `Dispenser` class and defining the following:

- a constructor taking an array of strings as its argument,
- the `isGlobal()` method, and
- the `getResourceAttributes()` method, which should return an instance of `ResourceAttributes` describing the resource managed by this isolate.

This third item hooks up the only two non-final classes of this API.

Most applications do not see dispensers, as they interact only with resource domains. Typically, only middleware, the JRE, or applications defining their own resources would explicitly create and register dispensers. Dispensers can also be created by passing their class name and constructor arguments to the virtual machine at startup time. This technique is appropriate for vital computational resources



**Figure 3.** Triggers and consume actions. The first and third actions are triggered by the consume request.

such as heap memory and CPU time. Dispensers created in this way are active throughout the entire lifetime of the virtual machine and cannot be unregistered.

A resource domain’s usage policy is specified as a set of actions containing pre-consume and post-consume callbacks that are invoked prior to and subsequent to a consume request. Whenever a client consume request is issued against its resource domain, this request is forwarded to the dispenser. The dispenser invokes the pre-consume callbacks to obtain the permitted number of units that the domain usage policy agrees to be consumed. The dispenser grants this amount if there are enough resource units available in the system and if the grant cannot lead to a violation of existing reservations. After making its decision, the dispenser invokes the set of post-consume callback actions for the domain and passes control back to the consuming client. (See Section 4 for a more detailed explanation.)

### 3.4 Consume Actions

The API defines a way to invoke an arbitrary *consume action* when a request to consume a given quantity of resource is made by an isolate bound to a resource domain. The following class encapsulates consume actions:

```
public final class ConsumeAction {
    public ConsumeAction(boolean persistent,
                          boolean synchronous,
                          ConsumeCallback callback,
                          Trigger trigger);
}
```

Actions can be persistent, in which case they remain in the system until explicitly removed. A non-persistent action is removed after the first time its callbacks execute. It can also be specified whether or not a post-consume callback is executed synchronously with respect to consume requests. Triggers and callbacks are the components of the consume actions that contain user-defined code. A trigger defines a condition upon which callbacks should execute, and the callback either decides whether further resource consumption is allowed or modifies an isolate’s behavior.

Triggers execute in the dispenser, whereas callbacks execute in the isolate that set the consume action. Triggers are invoked on each consume request and determine whether the associated callbacks will be executed. Figure 3 shows this relationship. Each trigger is connected to its corresponding consume callback with a solid line. For the illustrated consume request, some of the triggers fire and cause their callbacks to execute.

The next two subsections describe triggers and consume callbacks in more detail.

#### Triggers

Triggers are described by the following interface:

```
public interface Trigger extends java.io.Serializable {
    public boolean shouldFire(long current);
    public boolean shouldFire(long current, long proposed);
}
```

Triggers are executed in a dispenser and must be serializable so they can be transported between isolates. Their role is to act as gate functions determining whether an associated callback should be executed. The code executed within the

trigger could equally well be placed in the callback, but then an unnecessary round-trip inter-isolate communication would have to take place. Thus, triggers filter out callbacks that do not need to be executed upon a given consume/unconsume event.

The `shouldFire(long currentUsage, long proposedUsage)` method is invoked on each consume request against the dispenser in which the consume action has been installed. The return value determines whether the callback associated with the trigger will be invoked.

When a trigger is first installed, the dispenser invokes its `shouldFire(long currentUsage)` method. The rationale is to handle situations when the containing `ConsumeAction` is set after an interesting event has already taken place. For example, an application might be interested in being notified when memory usage exceeds 5MB, but usage is already 7MB at the time of setting the consume action.

By always returning true, a trivial trigger would never filter out any consume-related events and instead delegate all decisions to the associated `ConsumeCallback`. This choice would not impact application correctness, but could hurt scalability.

The following example trigger causes execution of its associated callback if the requested quantity would increase usage to at least the specified threshold:

```
public final class TriggerAbsoluteUp implements Trigger {
    private final long threshold;
    public TriggerAbsoluteUp(long threshold) {
        this.threshold = threshold;
    }
    public boolean shouldFire(long current) {
        return threshold <= current;
    }
    public boolean shouldFire(long current, long proposed) {
        return current < threshold && threshold <= proposed;
    }
}
```

During experimentation with the API, we found this trigger to be very useful and pre-defined it in the `Triggers` class, which is a factory for common triggers. `TriggerAbsoluteDown` is very similar; a subtle difference is that its `shouldFire(long)` method always returns false. The rationale is that, at trigger installation time, it cannot in general be determined whether the resource usage has ever crossed from above to below or equal to the threshold, which is a consequence of the fact that dispensers do not maintain a history of consume and unconsume requests.

Triggers comprising persistent consume actions can maintain information about previous consumption of the resource. For example, triggers that adjust the rate of consumption are useful in various situations. A typical rate adjusting trigger never enables its associated callback, but instead sleeps long enough so that the resource consumption rate does not exceed a desired value. In this way, rate adjusting triggers act as a throttle on the client's consumption.

Let us analyze a simple example of such a trigger. Its constructor takes two arguments, a threshold and a time

interval; their ratio is the desired upper limit on the rate of resource consumption. The `shouldFire(long)` method returns false because no information related to the rate of consumption is available to the trigger when it is installed. Whenever a consume request is made, the following happens:

```
// The time and current usage of the previous request.
long previousTime = -1, previousUsage = -1;
boolean shouldFire(long currentUsage, long proposedUsage) {
    if (previousTime != -1)
        record(previousTime, currentUsage - previousUsage);
    previousUsage = currentUsage;
    previousTime = currentTime;
    removeRecordsWithTstampsBefore(currentTime - interval);
    long amount = totalAmountRequestedInRecordedEvents();
    long delta = proposedUsage - currentUsage;
    if (amount + delta > threshold) {
        long interval1 = (interval * (amount + delta)) / threshold;
        Thread.sleep(interval1 - interval);
    }
    return false; // no need to invoke callback
}
```

Triggers are invoked as the first step of processing a consume request. A request may require evaluating multiple triggers, one for each associated action. Thus, the `shouldFire()` method above can know how much of the current request has been granted only the *next* time it is invoked. The first lines of `shouldFire()` are responsible for this: if this is not the first time the method is invoked, the time of the previous request must have been recorded along with the quantity granted, which is the difference between the usage then and now. Afterwards, records older than interval are removed, and the total amount of requested quantities is computed over the remaining records. If the amount increased by the quantity currently being requested exceeds the threshold, the trigger sleeps long enough to bring the rate of consumption down to the required range. As triggers operate within a critical section within the dispenser path, other potential consumers are held off during such a sleep. A useful variation on this theme is a trigger that adjusts the rate of requests themselves, not the rate of consumption.

## Callbacks

Consume callbacks are defined as:

```
public interface ConsumeCallback {
    public interface Pre extends ConsumeCallback {
        public long preConsume(ResourceDomain domain,
                               long currentUsage,
                               long proposedUsage);
    }
    public interface Post extends ConsumeCallback {
        public void postConsume(ResourceDomain domain,
                                long previousUsage,
                                long grantedUsage);
    }
    public interface PreAndPost extends Pre, Post {}
}
```

*Pre-consume* callbacks are executed prior to the dispenser's handling of the consume request. The `preConsume()` method has three arguments: the domain on which the consume request has been issued, the current usage, and the proposed usage – that is, the current usage increased by the requested amount, rounded up to meet granularity requirements. The value returned by `preConsume()` indicates to the dispenser how much of the request should be granted. A pre-consume callback that always denies the request would return `currentUsage`. Return values outside of the `[currentUsage, proposedUsage]` range are ignored. As multiple consume actions may be invoked on any consume, the dispenser needs to combine the return values of pre-callbacks. The default policy is to take the minimum, rounded up to the nearest granularity multiple. Finally, all pre-consume callbacks are executed synchronously with respect to the consume request and prior to it; otherwise, the return values would not be available to influence the dispensing decision.

Pre-consume callbacks and their triggers can be thought of as programmable constraints. In addition to denying the request, they can lower it or grant it unaltered, and, regardless of the outcome, they can modify the behavior of any isolate bound to the resource domain arbitrarily.

One operation that must be avoided in a callback is an attempt to consume or unconsume the resource in question, as this would cause deadlock. The remainder of the API's services is available for use within a callback.

In contrast to pre-consume callbacks, *post-consume* callbacks execute after the dispenser executes the triggered constraints and decides how much of the request should be granted. Post-consume callbacks can be viewed as notifications. They inform the isolate that set them about resource consumption decisions and allow for adjusting behavior to operate in changed conditions.

A callback may implement any of `ConsumeCallback`'s subinterfaces, including `ConsumeCallback` itself. (The latter case turns out to be useful in conjunction with rate-limiting triggers that never actually fire.)

There is no requirement that all consume actions to which a given isolate is bound be set by the same entity. Isolates can impose notifications on themselves so that they can react to triggered constraints. Privileged isolates can impose constraints on other isolates and thereby act as a resource manager for a set of isolates. 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 to lower its memory consumption and thus avoid violating a constraint.

In the following example, an isolate sender is bound by its creator to domains for CPU time and outgoing network traffic:

```
ResourceDomain.currentDomain(CPU_TIME).bind(sender);
ResourceDomain.currentDomain(NET_OUT).bind(sender);
```

The isolate can then use the RM API to be informed about excessive usage of either of the two resources and switch between two states: sending data in the uncompressed format if the most recent callback was caused by using more than 90% of the CPU time or sending data in the compressed format if the most recent callback was caused

by using more than 1MB/s of network bandwidth. Note that this code is very simple for presentation purposes.

// code in sender's main:

```
class ToggleCallback implements ConsumeCallback.Post {
    public void postConsume(ResourceDomain rd,
        long previousUsage,
        long grantedUsage) {
        String name = rd.getResourceAttributes().getName();
        if (name.equals(CPU_TIME))
            setCompressing(false);
        else if (name.equals(NET_OUT))
            setCompressing(true);
    }
}
```

```
ConsumeCallback callback = new ToggleCallback();
ResourceDomain.currentDomain(CPU_TIME)
    setConsumeAction(new ConsumeAction(true, false,
        callback, new RateDetectingTrigger(1000, 900))); // ms
ResourceDomain.currentDomain(NET_OUT)
    setConsumeAction(new ConsumeAction(true, false,
        callback, new RateDetectingTrigger(1000, 1 * MB)));
```

`RateDetectingTrigger` is a variant of the rate-adjusting trigger presented previously, and its arguments are the same: time interval and usage threshold. The difference is that rate detecting triggers do not adjust the rate but instead trigger the associated callback when the rate of consumption is too high.

### 3.5 Reservations

Reservations can be imposed on resource domains only if the underlying resource is reservable. An attempt to reserve a resource will fail if there is not enough of the resource to satisfy the reservation request. Reserving and consuming unbounded resources always succeeds unless constraints are imposed. Reservations are described by this class:

```
public final class Reservation {
    public Reservation(long value);
    public long getValue();
}
```

If setting the reservation succeeds (via `setReservation()` of `ResourceDomain`), the resource's implementation should be able to produce up to the reserved amount of the resource.

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 usage is high, giving them an opportunity to use less of a resource and avoid exceeding the reservation.

Reservations made on domains associated with the same dispenser should add up to, at most, a maximum value dictated by the underlying platform. Thus, setting a reservation may require lowering the reserved amount of another domain to succeed.

Initial reservations must respect resource limits imposed by the JRE or the underlying OS that the JRE must use. Corresponding to these initial reservations, implementations may impose "sentinel constraints" for a given resource to express these kinds of limits.

### 3.6 Resource Exceptions

The API defines two exceptions. Both are unchecked, to preserve source compatibility for existing RM-naive applications.

ResourceException is available for resource implementations to use in response to a failed call to consume(), to signal the absence of their resource to their end-clients. For instance, the consume() call for a file descriptor resource would be placed in the constructor of the java.io.File class. The constructor should return ResourceException if the underlying descriptor is not available, i.e., if consume() returns 0. However, resources with pre-existent failure semantics should continue to throw the corresponding exceptions. Thus, a resource manager for heap memory should throw OutOfMemoryException in preference to ResourceException.

The resource management framework classes generate ResourceManagementException when the implementation's integrity is compromised, for example, when an isolate in which a dispenser resides terminates while there are still resource domains bound to it. All public methods of the RM API can throw this exception.

## 4 ANATOMY OF CONSUME

To help clarify the ideas described above, this section presents the flow of control in consume. Figure 4 illustrates

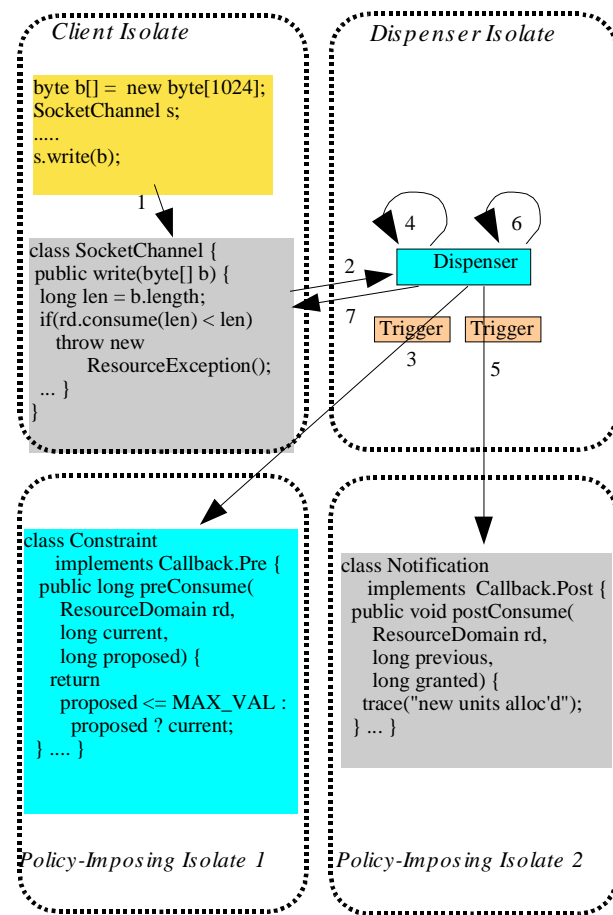


Figure 4. Flow of control in consume().

this control flow, with each arrow's label matching one of the steps in the description below.

**Step 1.** A client isolate requests  $x$  units of a resource, perhaps by invoking one of the resource's constructors. Control passes to the resource's implementation, which finds the client's resource domain and invokes consume( $x$ ) on that domain.

**Step 2.** The consume call is forwarded to the isolate that houses the dispenser, where the RM implementation acts upon the request. This code runs the set of triggers that are bound to the client's resource domain. A domain keeps a list of trigger and action pairs; an action's callbacks must be activated whenever its trigger evaluates to true. The output of this step is the set of callbacks that have to be invoked for the request. This set is denoted  $SC$  below.

**Step 3.** The implementation retrieves the current usage charged to the domain and from that calculates the new proposed usage, recording these values in `currentUsage` and `proposedUsage`. For each action in  $SC$  that has a pre-consume callback, the implementation instructs the isolate that set the callbacks's action to invoke that callback, supplying the domain, `currentUsage`, and `proposedUsage` as arguments. The implementation applies the dispenser's combining policy to merge the results returned from the callbacks, using `proposedUsage` rounded up by the resource's granularity if there were no callbacks. The resulting `proposedUsage` value becomes input to the next step.

**Step 4.** The dispenser now acts on the request, as modified by the callback invocations from step 3. The dispenser makes the final decision about how much of the request should be granted. This decision takes the resource's attributes into account. For instance, if the resource is unbounded, it will grant permission for the requested amount, since there is no inherent system limit on the number of resources. Similarly, if the resource is reservable, the amount will be granted only if the requesting domain has made a reservation for at least `proposedUsage` units. The value returned by the dispenser is `grantedUsage`.

**Step 5.** Once the dispenser returns its decision, the implementation reprocesses the action set  $SC$ , this time selecting actions that have post-consume callbacks. For each such action, the implementation instructs the isolate that set that action to invoke the callback, this time with the domain, `currentUsage`, and `grantedUsage` as arguments. Invocation is synchronous if the corresponding consume action was set with the synchronous property; otherwise, the callback is fired asynchronously. The order in which the callbacks are invoked is unspecified.

**Step 6.** Each non-persistent consume action in  $SC$  is removed from the set of callbacks tied to the domain.

**Step 7.** Control then returns to the resource's implementation in the client isolate, with the consume() method returning the value `grantedUsage - currentUsage`. This is the number of new resource units granted. The resource's implementation responds to this return value as appropriate for its semantics, throwing an exception if the quantity granted was insufficient to satisfy the client's original request. Finally, the client regains control, with its newly requested resource in hand or not, as the case may be.

## 5 DEFINING RESOURCES

The task of exposing resources through the RM API belongs to virtual machine implementers, implementers of the JDK, and to developers of libraries defining resources.

Consider a resource R. To make R manageable through the RM API, one must:

1. Define a concrete subclass of ResourceAttributes for R.
2. Define a concrete subclass of Dispenser for R.
3. Insert consume() and unconsume() calls where appropriate in the resource's implementation.
4. Apply optimizations.

The first two steps are straightforward. The crux lies in the third step, which covers the interaction between the resource's implementation and the RM API. The resource's implementation must consult the framework when handling requests to consume the resource, and the framework, after executing relevant callbacks, will advise whether the request should be granted. Beyond that, the interface does not specify in any way how resources should be implemented.

Optimizations are necessary to minimize the performance impact of controlling resource usage and are typically related to the resource's granularity. For instance, the implementation of a resource with a granularity of 1000 units might choose to buffer excess units when it expects to receive consume requests for a few units at a time.

Below, we illustrate how to expose resources through the RM API. Our examples include resources controlled by the JDK classes (the number of open sockets and the amount of data sent out over sockets), resources managed by the JVM's run-time system (heap memory), resources typically managed by the OS (CPU time), and a user-defined resource from a servlet engine.

The modifications described below concern the Multitasking Virtual Machine (MVM) [CD01], which is based on the Java HotSpot™ virtual machine [Sun00], and version 1.3.1 of the JDK.

### 5.1 Number of Open Sockets

Controlling the number of open sockets requires wrapping the actual socket creation code with invocations of RM API methods. To Java programs, sockets are instances of the java.net.Socket class, but the actual networking code is encapsulated in subclasses of the abstract SocketImpl class. We use the package-protected class java.net.PlainSocketImpl (the JDK's default socket implementation) as an example, because it is representative of the modifications necessary to control the number of open sockets. Its create() method includes an invocation of a private native method socketCreate(boolean stream) to associate the socket object with a socket created by a networking library of the underlying OS. All such invocations are bracketed by RM API code. (In the code below, SOCK\_NUM is a string naming the concrete subclass of ResourceAttributes that describes this resource):

```
// Throws ResourceException if the current isolate's not bound.
this.rd = ResourceDomain.currentDomain(SOCK_NUM);
if (rd == null) { // resource not registered in system
    socketCreate(stream);
    return;
}
```

```
if (rd.consume(1) != 1)
    throw new ResourceException("RM: can't open socket");
try {
    socketCreate(stream);
} catch (Throwable t) {
    rd.unconsume(1);
    throw t;
}
```

If the current isolate is not bound to a domain for the resource, the creation attempt fails immediately with a ResourceException. On the other hand, if the resource has not been registered with the RM implementation, creation is unconstrained. If the domain exists, it is recorded in the current instance of PlainSocketImpl, and an invocation of consume() requests one unit of the resource. If the request is denied, an appropriate exception is thrown. Otherwise, the socket is created; if that very last operation fails, the unit previously granted is returned back to the domain.

The pattern described above, which looks up the domain, asks it to allow the operation, and unconsumes the obtained quantity if the resource creation/allocation fails, is quite typical and applies verbatim to other resources such as the number of threads in use.

The clients of the original PlainSocketImpl could observe the following kinds of failure on an attempt to open a socket:

- lack of an appropriate Java security permission,
- lack of resources needed to create a socket, and
- bad arguments.

The new code accommodates the same failure modes. Since controlling open sockets as a computational resource is independent of their control as a security-sensitive resource, the lack of a permission still prevents socket creation. Bad arguments or lack of resources are still communicated to the applications as before. However, there is one new kind of failure: a ResourceException induced by a violation of the current resource domain's policy.

On UNIX® systems, a socket requires a file descriptor. Thus, should java.io be similarly retrofitted to make it RM-aware, socket creation could result in a ResourceException related to file descriptors and not to sockets.

It is important to reclaim sockets that are no longer in use. PlainSocketImpl is an example. Its close() method is called as a result of explicitly closing a socket and is also called by the finalizer to guard against forgetting to close the socket. We added the following code to close():

```
if (this.rd != null)
    rd.unconsume(1);
```

### 5.2 Amount of Data Sent over Network

Exposing the amount of data sent over the network as a resource manageable through the RM API requires modifying the sender. Consider java.net.SocketOutputStream as an example. This class's native socketWrite(byte[] buf, int offset, int len) method sends data on the socket associated with the given instance of the class. We wrapped the following code around the original invocation of socketWrite():

```
ResourceDomain rd =
    ResourceDomain.currentDomain(NET_OUT);
if (rd == null) { // resource is not registered; free rein!
```

```

    socketWrite(buf, off, len);
    return;
}
if (len > bufferedAmount) {
    bufferedAmount += rd.consume(len - bufferedAmount);
    if (len > bufferedAmount)
        throw new ResourceException("Net: can't send: " + len);
}
socketWrite(buf, offset, len);
bufferedAmount -= len;

```

In contrast to the number of open sockets resource, where the granularity was assumed to be 1, the granularity of the amount of network traffic must be much higher to avoid negative bandwidth and latency impacts from overly frequent invocations of `consume()`. Thus, the amount granted to consume requests but not used is buffered in the static variable `bufferedAmount`. When the buffered amount is insufficient, `rd.consume()` is called to replenish the buffer. Only after successful return from `socketWrite()` is the buffered amount decreased; if the bytes are not sent, the last line of the code fragment above is not reached. Thus, there is no need to return a granted, but unused quantity, because a subsequent invocation of `socketWrite()` will use it.

Combining the `NET_OUT` resource with the rate-adjusting triggers described in Section 3.4 yields a way of controlling network bandwidth. By having a trigger impose a suitable delay whenever a client application attempts to write through its sockets, a bandwidth manager can constrain the client's outgoing bandwidth.

### 5.3 CPU Time

As in many other JVM implementations, our system uses OS kernel threads to implement Java threads. The mapping is one-to-one. This suggests the following strategy for accounting for CPU time usage within an isolate: keep track of all the threads in the isolate and total up their CPU time usage when necessary, using available OS functionality to obtain this information.

We modified `java.lang.Thread` to have each new thread record its kernel thread identifier and add itself to a list of all threads in its isolate before executing the `run()` method.

A dedicated polling thread is responsible for CPU time usage accounting within each isolate. It wakes up at regular intervals and computes the usage to date. The polling thread is excluded from accounting, so that inactive isolates do not show spurious CPU time usage. The usage for the most recent polling interval is then reported to the current domain:

```

public void run() {
    long lastReportedUsage = 0;
    while (true) {
        Thread.currentThread().sleep(interval);
        long usage = getTotalCpuTimeUsage();
        long delta = usage - lastReportedUsage;
        if (delta == 0)
            continue;
        lastReportedUsage = usage;
        ResourceDomain rd =
            ResourceDomain.currentDomain(CPU_TIME);
        if (rd == null)
            continue;
    }
}

```

```

    if (rd.consume(delta) < delta) {
        error("No more CPU for: " + Isolate.currentIsolate());
        System.exit(1);
    }
}
}
}

```

Reporting only the usage over the last polling interval is a hardwired policy decision. A consequence is that whenever an already running isolate is bound to a domain for CPU time, accounting will not reach back to the very beginning of the isolate's execution.

Polling introduces fuzziness. In contrast to our previous examples, an isolate can use more of a resource than allowed. Such overuse can continue for up to the length of the polling interval. Similarly, reporting CPU time usage will not include the usage over the current interval. The user can learn about how much slack applications get, because the polling interval is defined to be the measurement delay (as recorded in the resource's attributes, Sec. 3.1).

### 5.4 Heap Memory

MVM's memory management is based on the carefully crafted and tuned code of the Java HotSpot virtual machine. It is vital not to compromise performance by adding RM API-related actions to any of the critical code paths. This requirement must be reconciled with the need to provide heap memory management with the correct resource characteristics: reservable, disposable, non-revokable, and bounded.

Heap organization in MVM is generational [Jone99]: each isolate has its own new generation (NG), while all of them share the old generation (OG). As each application needs its own NG, we define the granularity `G` to be NG's capacity. We decided to treat each isolate's dedicated NG as used memory, regardless of its actual occupancy. Thus, no accounting code need be interposed on allocations in NG, which is important since NG allocations tend to be very frequent.

There are two ways in which an object can find itself in OG: through a direct allocation (e.g., the virtual machine determines that a "large array" must be directly allocated in OG) or as a promotion from NG. In both cases, a per-domain counter is updated before the allocation. The counter is decremented upon OG collection, which is implemented as a four-phase mark-collect algorithm [Jone99]. During the first phase (marking), each isolate's memory usage is recomputed. This is cheap to do and unambiguous, as the garbage collection roots and objects of different isolates form disjoint sets.

Two facts imply that the binding between an isolate and its domain for heap memory is constant throughout the lifetime of the isolate. First, an isolate needs memory as soon as it is started. Second, an executing isolate needs memory and thus cannot be unbound (Sec. 3.2). Thus, maintaining per-domain counters requires a simple map of constant entries associating isolates with domains.

A helper class is loaded into each isolate. The memory management subsystem calls this class whenever an isolate's usage of OG would cross a granularity boundary. Up-called methods in the helper class are very simple: they invoke

consume() (before granularity-crossing allocations in OG) and unconsume() (after granularity-crossing collections) on the current domain for heap memory. If insufficient memory is granted to a consume request, the corresponding allocation results in `OutOfMemoryError`.

Promotions happen during *garbage collection safe-points*, when no allocations can take place. Executing a callback action may, in general, require creation of new objects and thus cannot be performed during a safe-point. Therefore, if a promotion could cause an isolate whose NG is currently being collected to increase usage of OG so as to cross a granularity boundary, promotions are stopped. As a consequence, some objects may have to wait one NG collection longer before promotion, but this is transparent to Java programs.

The helper class is up-called after NG collections during which promotions were stopped. These up-calls are executed before the allocation attempt that caused the collection returns. If the up-call's request for more OG space has been turned down, the allocation can succeed only if there is enough space for it in NG.

This approach has several desirable properties. First, failures are always linked to allocations, so memory shortage can be signaled to applications as an allocation error. Second, an inexpensive implementation is possible, because no critical code path is lengthened. Third, as no up-calls to the helper class happen when in a safe-point, callbacks triggered by their `consume()` and `unconsume()` invocations can create objects or even invoke `System.gc()`. Fixing G at NG's capacity is acceptable, because this is an implementation-dependent resource attribute, and G reflects a natural unit of management for this particular heap organization.

## 5.5 Servlet Resources

We experimented with the RM API's interaction with a web server modified so that each servlet runs in its own isolate. An example of a resource useful in this context is *load*, defined to be the sum of weights assigned to a collection of executing servlets. Weights are assigned off-line and are stored in deployment descriptors. Load is a non-traditional resource that can be easily expressed in our framework. We found that it was easy to express non-trivial policies, such as a model in which services are free up to the point when contention for them starts. When the load exceeds the limit, all clients for which servlets are currently running are billed in proportion to their share of the overload.

Exposing servlet load as a resource through the RM API included the step of defining subclasses of `Dispenser` and `ResourceAttributes`. The resource is non-reservable, unbounded, non-revokable (one cannot remove instances of that resource without the client failing), and disposable (since the termination of a servlet request results in freeing up the load units consumed by that servlet).

The `consume()` method is invoked on the resource domain each time that a servlet `doXXX()` method is called, and an `unconsume` request is issued following the method's termination. The following code segment has been inserted into the web server's code that dispatches `doXXX()` methods. It calculates the weight of the servlet. If the consume is permitted, the servlet weight is added to the current load.

```
ServletConfig config = ... /* configuration object for servlet */;
// Get the weight and consume it.
int weight = getServletWeight(config);
if (consume(weight) < weight)
    throw new ResourceException("Server load exceeded");
addWeightToCurrentLoad(weight);
```

Similarly, upon servlet termination, its weight is subtracted from the current load.

## 6 PROTOTYPE IMPLEMENTATION

We implemented the resource management framework on top of the Isolate API. Our prototype did not require any changes to the virtual machine and is coded entirely in the Java programming language (6K lines of source code, 170KB of bytecode). However, exposing a given resource through the API may require changes to its implementation, which, in turn, may require changes to the JVM or JDK; see, for example, Sections 5.1-5.4.

A consequence of not modifying MVM is that a supplementary non-public class, `Environment`, wraps around each isolate to manage information related to the RM API. An isolate's environment stores the names of the resource domains to which the isolate is bound, a link to the dispenser isolate for each of these domains, local dispensers that the isolate may have created, and so on.

Another helper class, `BootstrapEnvironment`, executes as the first isolate. This class reads a configuration file containing the names of dispensers that must be available at start-up time and then instantiates them. Thus, the first isolate hosts dispensers deemed vital; it does not host any applications and is thus insulated from their failures. After this setup, `BootstrapEnvironment` creates another isolate, which executes the main application code. Before being started, this second isolate is bound to a domain for each dispenser already created; each such domain has its reservation set to maximum if its resource is reservable. As Section 3.5 explains, if the second isolate needs to create other resource domains with reservations, it must correspondingly lower its own reservations.

`BootstrapEnvironment` maintains a map of resources to dispensers. When an isolate is created, its `Environment` sends the list of resource domains to which the isolate is bound to `BootstrapEnvironment`, which replies with the list of links to *peer objects*, where a peer object represents a resource domain in a dispenser. Operations on resource domains are forwarded on to their peer objects. A pair of links is used for this purpose: one handles consume requests, and the other handles the remaining domain operations. This design permits non-consume resource domain operations to be executed within synchronous callbacks.

### 6.1 Performance

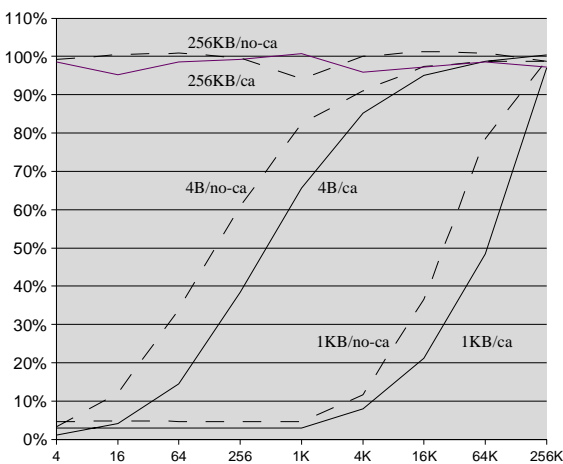
The preliminary performance data presented below was obtained using the prototype described above running on MVM on the Solaris™ 9 Operating System on a Sun Enterprise™ 3500 server with four UltraSPARC™ II processors and 4GB of main memory.

The implementation has the property that when the RM API is not used, applications do not incur any cost. When it is used, the cost in most circumstances is zero or negligible

and usually depends on the resource granularity. For example, the length of the polling time interval in CPU time management (Sec. 5.3) determines the overhead to applications. For intervals of 100 milliseconds or more, there is no measurable overhead for any of the SpecJVM98 benchmarks [Spec98]. A 50ms interval results in an average overhead of 0.7% while a 10ms interval leads to an average overhead of 3%.

Using the RM API to control the amount of data sent over the network exhibits a similar relationship between granularity and cost (Fig. 5). In this experiment, two isolates, executed in the same instance of MVM, repeatedly exchange a byte array over a local socket, with both isolates bound to the same resource domain for outbound network traffic. The parameters of the experiment are the resource granularity (X axis in the figure), the size of the byte array (the number in the legend), and whether consume actions are associated with consume requests (the *ca/no-ca* item in the legend). If a consume action is set, it is non-persistent, and the trigger always enables the action's callback, which grants the request and then re-establishes a new consume action to be triggered when usage exceeds another granularity point. The plots show the bandwidth achieved at a given byte array size when the RM API is enabled relative to bandwidth when it is disabled.

The top two plots (array size 256KB) indicate that for large sizes of individual transfers, the overhead of the RM API is negligible, because consume requests are executed relatively infrequently; the actual data transfer takes up most of the execution time. For very small arrays (4B; two lines in the middle of the figure) and small granularities, the RM API's overhead can be substantial. For larger granularities, buffering of granted but not consumed amounts pays off. For a 1KB array size, the bandwidth drop is very high for small granularities. This dropoff is explained by the fact that 1KB transfers deliver much higher bandwidth than 4B transfers, and applying the RM API slows 1KB transfers much more than the already slow 4B transfers. Overall, for granularities of 256KB or higher, the cost of the RM API is not noticeable. The additional overhead of consume actions is noticeable, but also disappears at large granularities.



**Figure 5.** Relative impact of the RM API on network bandwidth, as a function of granularity.

For all but two of the SpecJVM98 benchmarks, heap management, as described in Sec. 5.4, does not introduce measurable overheads when the granularity (and, consequently, new generation capacity) is at least 2MB. For *javac*, the overhead is about 3% and for *jack*, about 1.5%. Object allocation and tenuring patterns for these two benchmarks lead to more frequent invocations of `consume()` than it is the case for the other tests.

## 7 DESIGN DECISIONS

In this section, we discuss several design decisions and their implications.

### 7.1 Why Isolates?

We use isolates to represent computations because it is more convenient to manage resource-consuming programs that do not directly share resources. It makes accountability simpler, since any given unit of a resource has only one owner. Alternatives to isolates as units for controlling consumption include threads and class loaders. However, neither really works for this purpose, due to difficulties related to designing predictable, unambiguous, and easy to understand rules and mechanisms associating resources with their users. A thread can traverse many program components, and resources obtained by one thread can get referenced and consumed by other threads. Answering seemingly trivial questions such as, “How much memory is this thread using?” becomes imprecise. Depending on the definition chosen for “owning a resource,” accounting in the presence of sharing is likely to lead to the following:

- *over-accounting.* For example, the sum total of all memory “used” by each thread exceeds the amount of allocated data.
- *under-accounting.* For example, a resource allocated by a thread that has already exited is used by another thread, and the accounting subsystem does not keep track of the ownership change.
- *confusing accounting.* For example, one does not expect resources used by a suspended thread to change until the thread resumes; in accounting models where each sharing party is charged  $1/n$ , this property does not hold.
- *expensive accounting.* Trying to compensate for the problems above invariably requires elaborate accounting schemes.

We are convinced that making threads units of resource management is, in general, an incorrect level of accountability. If it is important to make sure that a particular thread's resource consumption is controlled, the thread must be the only one in an isolate that is exclusively bound to domains for the resources of interest.

Having class loaders represent the unit of accountability has similar problems, since class loaders do not isolate one application's data from another's.

### 7.2 Security

The interface offers a high degree of control over quantities of resources available to computations. Some operations are potential security hazards – for example, binding an isolate to a CPU time domain with a denying constraint set to 0ms effectively prevents the isolate from executing. Security is an important issue, since control of vital computational

resources, like heap memory or CPU time, must remain under the control of trusted isolates. Appropriate security permissions are required in many places to guard the use of the API. One example is setting a synchronous callback action which, maliciously exploited, could lead to a domain being blocked due to a non-returning callback. Another is the installation of a trigger that does not belong to the pre-defined set of triggers since, maliciously exploited, trigger code could be used to leak or corrupt data in the dispenser's isolate.

We have implemented a version of RM API with security checks in place. This version does not change the existing Java 2 platform security model [Gong99], nor is the API associated with the standard security permissions. It is a programmatic model in the sense that all operations involving the creation and delegation of privilege in the RM framework are coded as methods of the RM API, rather than done by using the isolate configuration files. The basic model used is to grant the privilege to create a resource domain to the dispenser isolate for that resource. An isolate can create a domain only if it has been explicitly granted that privilege from the dispenser isolate. Similarly, only a resource domain creator possesses the privilege to associate synchronous callbacks or to bind isolates to the domain although the domain creator isolate can grant that privilege to another. The goal was to ensure compatibility with the Java security model. Finalizing the details is underway.

It is important to note that the API does not change the existing Java 2 Platform security model. However, use of the RM API does introduce a certain division of responsibility. For instance, the framework might permit an isolate thread to create a resource, yet the creation might be refused by the security policy. This can happen because the security policies and resource management policy are not necessarily specified by the same authority. Similarly, 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 domain 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 RM API provides a controlled and uniform way of dealing with such situations. The RM 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.

### 7.3 Hierarchies

The RM API relies on a hierarchy of isolates for local dispensers. Whenever an isolate creates a resource domain, the RM infrastructure looks in the current isolate for a dispenser. If none is found, the framework looks in that isolate's parent, and then its parent, and so on. This RM design choice reflects the hierarchy that often exists in

systems. For instance, a system isolate may create a servlet engine inside of a new isolate that, in turn, creates servlets inside of further isolates. In such situations, it is logical for the dispenser for servlet resources to be located in either the system or servlet engine isolates.

No explicit notion of hierarchies among resource domains is provided. Grouping or nesting of domains has to be programmed explicitly. For example, to maintain knowledge about combined usage of a set of domains for the same resource, one would have to rely on notifications. We are contemplating facilitating such operations through the RM API.

## 8 RELATED WORK

Resource management was pioneered by the early multi-programmed and time-sharing operating systems. The resources in question were basic ones such as CPU, network bandwidth, disk space, and main memory. There has been much less work in resource management for the Java platform. JRes [CvE98] is an early proposal of resource control interface for Java applications. The interface is not extensible and enables accounting for and limiting the usage of the CPU time, network bandwidth, and heap memory usage. The unit of accounting is an individual thread, and there is no notion of an application as an isolated execution entity, which leads to problems outlined in Sec. 7.1. The focus of JRes was primarily on how accounting can be implemented without modifying the virtual machine. The techniques used for this purpose included bytecode editing (off-line for core classes, on-line for application code) and native code.

J-SEAL2 [BHV01] also focuses on how to account for resource consumption, and the API is similar to this of JRes. J-SEAL2 performs all accounting in a portable way, through bytecode editing. All methods are rewritten to include calls to monitoring objects that register the memory and CPU consumption of the seal. Memory is measured by having the size of each object created (calculated from the number of object fields) added to the memory accounting object on each `new()` operation. CPU is estimated through the number of byte-code instructions executed.

In J-SEAL2, control is applied to *seals* [BV99] and not to individual threads. Seals encapsulate protection domains, with their own threads and class loaders, and do not allow cross-seal sharing. The J-SEAL2 system contains an implicit resource management policy that stems from its protection model. The set of seals is organized as a hierarchy. The members of the initial set of application seals are each attributed a heap size and percentage of CPU cycles. Whenever a seal creates a child seal, the parent can only grant to the child resources that it has been attributed.

The goal of the interface presented in this paper is much broader in scope than JSEAL-2 or JRes. We seek to allow application programmers to define their own resources, as well as manage core resources such as those of the JDK classes. Our API does not dictate any RM policy but instead allows applications to design their own via flexible mechanisms. The RM API can be used with widely differing resource management implementations, and thus resource management overheads can be minimal. In contrast, both JRes and J-SEAL2 use bytecode editing as their primary means of

implementing resource management, which results in overheads on the order of 20%.

KaffeOS [BHL00] implemented the process abstraction within the JVM so that mistrusting programs can co-exist within the same underlying address space. The JVM is modified to prevent direct object sharing between process spaces, although shared heaps are provided for efficient sharing between trusted processes. The JVM can account for each process's usage of CPU and of memory, as well as for the time spent by the garbage collector on behalf of each process. The purpose of the resource accounting is to extend the protection model to detect and thwart denial of service attacks launched by malicious programs. KaffeOS focuses on CPU time and heap memory and is not designed for application-specific resources. The goals of the Aroma virtual machine [SBB+01] included the ability to control resources such as disk and network bandwidth and CPU time. CPU time usage can be expressed either as a number of bytecodes executed during a time interval or as a percentage of processor time.

Overall, we are not aware of an operating system or a virtual machine modification, either commercially available or research, whose interfaces to control resource consumption are as flexible and extensible as the ones proposed in this paper. Even OSes based entirely on safe languages, such as JX [GFW+02], do not offer RM APIs matching the level of abstraction of modern programming languages.

The Real-Time Java Specification (RTJS) [BGB+00] defines abstractions that are compatible with the standard JVM and that permit applications with real-time constraints to be run. These abstractions concern thread scheduling, memory management, and event handling. Controlling thread scheduling is necessary to allow high priority tasks to complete on time. The RTJS defines *scheduler* objects that are used to control (e.g., set priorities, or preempt) real-time threads objects whose semantics are also defined by the framework. The issue for memory management is preventing the garbage collector from interfering with the execution of threads so that their execution times can be made more predictable. The framework defines several types of memory regions that can be shared by threads, where the lifetime of objects allocated within these regions is determined from the visibility of the region and not of the objects. For example, when the last thread leaves a *ScopedMemory* object, all objects allocated in that memory object can be reclaimed. The memory object interface also permits programs to learn about the current consumption of that memory resource.

The event handling mechanism in RTJS has similarities to the callback mechanism in the RM API. An *AsyncEvent* object represents any event that can be asynchronously generated from the program's environment. Any number of *AsyncEventHandler* objects can be bound to any instance of an *AsyncEvent* object. A handler is a thread that is started when any event that it is bound to is fired. Any number of handlers may be bound to the same event; the handlers are executed serially when the corresponding event is fired. A handler may be bound to several events.

Despite the similarity of mechanisms with respect to event handling, there is no relationship between the RTJS and the

RM API. RTJS is specialized for real-time platforms, with a fixed set of resources and strict timeline guarantees requirements, while the RM API's goals include extensibility, flexibility, and cross-scenario applicability that are likely to appeal much more to non-RT programmers.

## 9 CONCLUSIONS

This paper describes a resource management interface for the Java platform. The distinguishing features of this proposal are its extensibility, flexibility, completeness of abstraction, wide applicability, and compatibility with existing applications. The interface insulates users from implementation details such as whether isolates are processes or computations within the same virtual machine and whether a given resource is managed by the OS, the JVM, the core JRE classes, or by an application. As well as accommodating traditional resources such as heap memory and CPU time, the interface allows application writers to define resources that are meaningful to their problem domains and to manage their applications with respect to those resources.

We have prototyped the interface entirely in the Java programming language. Modifying the virtual machine for that purpose can be more efficient, although the performance achieved with the current prototype is already satisfactory for practical use. To a large extent, this is due to the fact that the interface allows for managing tradeoffs between the precision of resource accounting and its cost.

Retrofitting existing resource implementations to take advantage of the RM API is relatively easy, as it only involves appropriately inserting consume and unconsume requests. We were also pleased with the feel of using the interface – in our opinion, it matches the level of abstraction of the language. This is not surprising – the proposal is a distilled result of numerous discussions and lessons learned from designs that just did not feel right. We hope that the RM API will become a seed for a standardization effort and, consequently, will be found equally useful by many other people.

We have done our prototyping in the context of the Java platform, but we believe that our key design decisions and resulting abstractions are applicable to other safe language environments, as well as to constructing complete operating environments in general.

**Acknowledgments.** The authors acknowledge valuable input from Walter Binder, Greg Bollella, Bill Foote, Mick Jordan, Hideya Kawahara, Tim Lindholm, Doug Lea, Gary Pennington, Niranjan Suri, Pat Tullmann, Jan Vitek, Mario Wolczko, and the JSR 121 Expert Group. Special thanks are due to Laurent Daynes for his help with the MVM's implementation of the JSR 121. Much of this work was done while Ciarán Bryce was visiting Sun Microsystems Laboratories from October, 2002, through March, 2003. The authors thank the Computer Science Department of Geneva University for allowing this visit.

## 10 REFERENCES

[BHL00] Back, G., Hsieh, W., and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4<sup>th</sup> OSDI, San Diego, CA, 2000.

- [BHV01] Binder, W., Hulaas, J., and Villazon, A. *Portable Resource Control in Java: The J-SEAL2 Approach*. 16<sup>th</sup> ACM OOPSLA, Tampa Bay, FL, October 2001.
- [BGB+00] Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Hardin, D., and Turnbull, M. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [BV99] Bryce, C. and Vitek, J. The JavaSeal Mobile Agent Kernel. 3<sup>rd</sup> International Symposium on Mobile Agents, Palm Springs, CA, October 1999.
- [CvE98] Czajkowski, G., and von Eicken, T. *JRes: A Resource Accounting Interface for Java*. 13<sup>th</sup> ACM OOPSLA, Vancouver, BC, October 1998.
- [CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution*. ACM OOPSLA'01, Tampa, FL.
- [GFW+02] Golm, M., Felser, M., Wawersich, C., Kleinoder, J. *The JX Operating System*. The USENIX Annual Technical Conf., Monterey, CA, June 2002.
- [Gong99] Gong, Li. *Inside Java 2 Platform Security*. Addison Wesley, 1999.
- [GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. 2<sup>nd</sup> Ed. Addison-Wesley, 2000.
- [Hall00] Hall, M. *Servlets and JavaServer Pages*. Prentice Hall, 2000.
- [JCP01a] Java Community Process. JSR 108: Units Specification API. <http://jcp.org/jsr/detail/108.jsp>.
- [JCP01b] Java Community Process. JSR 121: Application Isolation API. <http://jcp.org/jsr/detail/121.jsp>.
- [Jone99] Jones, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1999.
- [SBB+01] Suri, N., Bradshaw, J., Breedy, M., Ford, K., Groth, P., Hill, G., and R. Saavedra. *State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine*. Java Virtual Machine Research and Technology Symposium, Monterey, CA, April 2001.
- [Spec98] Standard Performance Evaluation Corporation. *SPEC Java Virtual Machine Benchmark Suite*. August 1998. <http://www.spec.org/osg/jvm98>.
- [Sun00] Sun Microsystems, Inc. *Java HotSpot™ Technology*. <http://java.sun.com/products/hotspot>.

## 11 APPENDIX: ISOLATE ESSENTIALS

The resource management interface proposed here considers isolates as the program units whose consumption is being controlled. An isolate is a Java application component that does not share any objects with other isolates. Isolate creation and life cycle management are the subject of the Application Isolation API (“Isolate API”), the formal output of JSR-121. Since isolates form an important part of the RM API, we give a brief overview here. The interested reader is referred to [JCP01b] for more information.

The Isolate API allows for the dynamic creation and destruction of isolates. The creating isolate may pass contextual information to its offspring that is required by the application started within the new isolate. Isolates do not share objects, but they can communicate using traditional inter-process such as sockets and files. The Isolate API also

defines an inter-isolate communication mechanism known as *links* for synchronous and unidirectional message passing. Isolates may exchange instances of `java.io.Serializable` as well as isolate and link references over links. The ability to send links on links is crucial because there is no central registry of isolates and their communication endpoints – any topology needed by applications must be created. Links have rendezvous semantics – the sender is blocked until the message is received. Bootstrapping communication is achieved by passing links to the `start()` method of an isolate. In the following code extract, a class `Start` creates two isolates, `Ping` and `Pong`, and installs a link from the former to the latter. This link is passed as a parameter to the `start` methods of the two isolates. `Ping` sends a message on this link, which `Pong` reads and prints on its output stream.

```
public class Start {
    public static void main(String[] args) {
        Isolate ping = new Isolate("Ping", new String[0]);
        Isolate pong = new Isolate("Pong", new String[0]);
        Link pingToPong = Link.newLink(ping, pong);
        pong.start(new Link[] { pingToPong });
        ping.start(new Link[] { pingToPong });
    }
}

class Ping {
    public static void main(String[] args) {
        Link out = Isolate.getLinks()[0];
        out.send(LinkMessage.newSerializableMessage("ping"));
    }
}

class Pong {
    public static void main(String[] args) {
        Link in = Isolate.getLinks()[0];
        System.out.println(in.receive().getSerializable());
    }
}
```

The Isolate API is 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. The Isolate API lends itself to different implementation strategies. One approach is to exploit the OS process boundaries to implement the isolate boundaries. In this case, an application resides within a single JRE, and a set of process-based JVMs cooperate to implement the whole JRE infrastructure. An alternative implementation strategy is for all isolated applications to reside within a single JRE instance and for the JRE to provide support to implement the protection boundaries within a single address space. This approach was used to implement the Isolate API built on top of the Multitasking Virtual Machine. The key point is that whatever the implementation strategy chosen, the behavior of the isolate application remains the same.

## ABOUT THE AUTHORS

**Grzegorz Czajkowski** is a Senior Staff Engineer at Sun Microsystems Laboratories, which he joined in the summer of 1999, after obtaining a Ph.D. in Computer Science from Cornell University. He is interested in operating systems, programming languages, and middleware. Currently he is leading the Barcelona project, which focuses on architectures for improving the scalability and reliability of the Java platform.

**Stephen Hahn** is a Senior Staff Engineer in the Solaris Kernel Technologies group at Sun Microsystems. His recent work has been focused on resource management at the operating system level, particularly in building foundations for automated resource managers. His research interests are broad, and include understanding source code community formation, describing meaningful application interdependencies, and implementing high performance sort algorithms. He received his Ph.D. in Theoretical Physics from Brown University, before joining Sun in 1997.

**Glenn Skinner** is a Senior Staff Engineer at Sun Microsystems Laboratories. His research interests focus on the confluence of operating systems and the Java Platform. He is currently a member of the Barcelona project, which focuses on architectures and interfaces aimed at increasing the scope and scalability of the Java platform.

Before moving to Sun Labs, he spent many years at Sun as an operating system developer, contributing to many areas of the Solaris kernel, including STREAMS, the virtual memory subsystem, and, most significantly, the file system infrastructure. After a short detour into the Sun Cluster group, he became a JVM implementer, which forced him to notice the similarities between operating systems and Java Virtual Machines and sparked his interest in fostering their harmonious coexistence.

**Pete Soper** is a Staff Engineer in the Sun Microsystems Java Software engineering group and the Specification Lead of JSR-121, driving a Java process abstraction into that platform. His interests include operating environments and language systems and the open source software movement. Before joining Sun he explored human factors related to NASA manned space missions, created components of a portable operating system and realtime executives for communication products, and developed third generation language products for SMP systems at Encore Computer. He holds a bachelors degree from the University of Alabama.

**Ciarán Bryce** is Assistant Professor at the University of Geneva, Switzerland. He obtained a computer science degree at Trinity College Dublin in 1991 and a Ph.D. from the University of Rennes in 1994. He worked as a researcher at INRIA-Rennes (in France) from 1991 to 1994 and at GMD (German National Research Centre for Information Science) from 1994 to 1997. He has been with Geneva University since 1997. His research interests include computer security, object-orientation, and more recently, wireless information systems.