Extending a J2EE™ Server with Dynamic and Flexible Resource Management

Mick Jordan[1], Grzegorz Czajkowski[1], Kirill Kouklinski[2], Glenn Skinner[1]

[1]Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054, USA
{firstname.lastname}@sun.com
[2]School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada
kkouklin@math.uwaterloo.ca

Abstract. The Java™ 2 Platform, Enterprise Edition (J2EE™) is the standard platform for hosting enterprise applications written in the Java programming language. A single J2EE server can support multiple applications much like a traditional operating system, but performance levels can be difficult to control, due to the absence of resource management facilities in the Java platform. The Resource Management (RM) interface addresses this problem by providing a flexible and extensible framework for managing resources that is applicable across a broad spectrum, from low-level resources like CPU time to higher-level resources such as database connections. RM has been implemented in the Multi-tasking Virtual Machine (MVM), a scalable operating environment for multiple applications based on the concept of isolated computations. This paper describes the application of MVM and RM to the management of resources in a J2EE Server and shows that application performance can be controlled flexibly and easily with low overhead and minimal intrusion.

1 Introduction

The Java™ 2 Platform, Enterprise Edition (J2EE™) [1] is the standard server-side environment for developing enterprise applications in the Java programming language. J2EE is itself layered on the Java™ 2 Platform, Standard Edition (J2SE™) [2]. In many respects the combination of the J2EE and J2SE platforms subsumes the underlying operating system (OS). For example, a single J2EE server can host several applications, possibly from different organizations, that must compete for the server resources. Many of the traditional OS constructs are overlaid or replaced with J2SE or J2EE counterparts. For example, the J2SE thread model subsumes the native OS thread model. Similarly, a J2EE *container*, which hosts a component of a J2EE application, shares some characteristics with an OS process. Both developers and administrators interact mainly with the APIs provided by J2EE/J2SE and only experience the underlying OS APIs through the filters provided by J2EE/J2SE. The main benefit is the portability of the J2EE/J2SE platform between different operating systems.

In practice, however, the J2EE/J2SE platforms omit some important features that are standard in operating systems. In particular, unlike OS processes, J2EE applications cannot be properly isolated from one another and many resources required for adequate application performance cannot be controlled with platform facilities. These deficiencies require administrators to bypass the J2EE/J2SE layers and interact directly with the native OS APIs, where available, thus limiting overall portability.

Consider, for example, a J2EE server that has two different applications deployed and executing. These applications are similar to processes in an OS environment. Therefore, it should be possible to control the resources available to each application using operations analogous to those available for processes, for example, controlling the amount of memory or CPU. In current J2EE platforms this capability is notably absent, essentially because the underlying J2SE platform does not provide controls on these familiar resources. Furthermore, the J2EE platform defines additional resources, with no obvious analog in the operating system context, for example, the number of JDBC™ connections available to an application. Currently, J2EE servers provide some ad hoc mechanisms for controlling such resources, but it is not easy to partition them between separate applications within a single server.

The approach most J2EE server vendors take to solve this problem is to directly exploit the underlying OS facilities by exposing these to the J2EE administrator. Essentially, this involves mapping each J2EE application to a distinct server instance, so that the process-based resource management mechanisms of the OS can be utilized. One consequence is that the system's overall resource consumption is increased, due to the increased number of processes. Another is that part of the API cannot be expressed in terms of the Java programming language. Although this can be papered over to some extent by management frameworks and tools, e.g., Java™ Management Extensions (JMX ™)[3] and CIM [4], the end result is more complexity and lower performance than would be possible if the facilities were provided directly in the J2EE/J2SE platform.

In previous work [5] we described a flexible and extensible framework for resource management that is expressed entirely in the Java programming language and is capable of efficiently handling all the traditional resources, such as CPU time, memory, and network, as well as programmer-defined resources such as JDBC connections. This work was itself built on earlier work that developed a programming model, and an associated API extension to the Java platform, that supports fully isolated computations. The API development, carried out as a Java Specification Request under the Java Community Process, is described by JSR 121 [6], and introduces the *isolate*, which abstracts the notion of a program in the Java platform, without prescribing a particular implementation. Resource management (or RM) is based on isolates in that an isolate is the fundamental unit of accounting. This solves many difficult problems relating to resource sharing and ownership. We have built a prototype implementation of RM as an extension to the Multi-tasking Virtual Machine (or MVM) [7] that implements isolates within a single Java™ Virtual Machine (JVM™).

In [8] we discussed how isolates might be utilized within a J2EE server, specifically in the context of the J2EE 1.3.1 Reference Implementation (J2EERI) [9]. A key concept was an *application domain*, that encapsulates an entire J2EE application, and is represented using one or more isolates. The main rationale was based on being able to exploit RM to provide J2EE-application-centric resource management.

This paper addresses the applicability of RM to a J2EE server, in the context of J2EERI. We describe how we integrated RM into the J2EERI and provide some initial measurements of its ability to effectively manage resources in a multi-application environment.

The rest of the paper is structured as follows. Section 2 provides an overview of the MVM architecture and the isolate programming model. Section 3 describes the RM framework. Section 4 contains a very brief overview of J2EE and explains how we applied MVM and isolates to J2EERI. Using J2SE resources and adding new, J2EEspecific ones, to J2EERI is described in Section 5. A summary of related work, future work, and conclusions complete the paper.

2 Background on Isolates and the Multi-tasking Virtual Machine

An isolate is a Java application component that does not share any objects with other isolates. The Isolate API [6] allows for the dynamic creation and destruction of isolates. A simple example of the API is the creation of an isolate, that will execute MyClass with a single argument "abc":

```
new Isolate("MyClass", new String[] {"abc"}).start();
```

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 communication mechanisms 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 – applications must explicitly create any topology they need.

The Isolate API is fully compatible with existing applications and middleware. In particular, applications unaware of the API may be managed as isolates without need for modification. Likewise, middleware can be unaware of or ignore the API.

Our implementation extends the rudimentary life-cycle management features of the Isolate API with the ability to suspend and resume isolates. We found this addition particularly useful when enforcing CPU time consumption policies (Sec. 5.1).

The Isolate API lends itself to various implementation strategies. One approach is to exploit the abstraction of an operating system process to implement the isolate boundaries. An alternative implementation strategy, employed in the design of MVM, is for all isolates to reside within a single instance of the Java™ runtime environment (JRE) that implements the protection boundaries within a single address space[7].

MVM is a general-purpose virtual machine for executing multiple applications written in the Java programming language. It is based on the Java HotSpot™ virtual machine (HSVM) [10] and its client compiler, version 1.3.1 for the Solaris™ Operating Environment.

MVM aggressively and transparently shares significant portions of the virtual machine among isolates. For example, run-time representations of all loaded classes and compiled code are shared, subject to modifications that prevent inter-isolate interference. Runtime modifications make the replication of non-shareable components (e.g., mutable state of classes) transparent. In effect, each application "believes" it executes in its own private JVM, as there is no inter-isolate interference due to mutable runtime data structures visible directly or indirectly by the application code. Certain JRE classes, such as System and Runtime had to be modified to make operations such as System.exit() apply only to the calling isolate. The heaps of isolates are logically disjoint, since the required level of isolation implies that isolates cannot share objects.

In MVM, each isolate transparently benefits from the class loading and method compiling work done by other isolates. This provides a significant reduction in the startup time and memory footprint. Execution time also benefits due to co-locating frequently communicating programs in the same OS process [8].

MVM operates as follows. The first isolate is a simple application called *Mserver* that listens on a socket for connections. The *java* command invoked by the users is replaced with the *jlogin* program, written in C, that connects to *Mserver* and passes to it the command line arguments (the main class name and its arguments, *-D* flags, etc.). *Mserver* creates a new isolate according to the obtained request. Standard input/output/error streams are routed to the *jlogin* process that launched the isolate.

3 Overview of the Resource Management API

This section contains a brief overview of the main features of the RM API [5]. It is important to note that existing applications can still run without modification, even if the JRE classes they depend on exploit the RM framework. Applications that need to control how resources are partitioned (e.g., application servers) can use the API for that purpose. Pro-active programs can use the API to learn about resource availability and consumption to improve the characteristics most important in the given case (response time, throughput, footprint, etc.) or to ward off denial of service attacks.

Resources can be exposed through the RM API in a uniform way, regardless of whether they are actually managed by the operating system (e.g., CPU time in JVMs that rely on kernel threading), the run-time system (e.g., heap memory), core classes (e.g., file and network resources), middleware (e.g., JDBC connections), or by the application itself. Retrofitting existing resource implementations to make them exploit the RM API is relatively easy, and the trade-off between the cost and precision of usage accounting is programmable [5].

The unit of management for the RM API is an isolate. This choice makes accountability unambigous, as each resource in use has exactly one owner.

A *resource domain* encapsulates a usage policy for a resource. All isolates *bound* to a given resource domain are uniformly subject to that domain's policy for the underlying resource. An isolate cannot be bound to more than one domain for the same resource, but can be bound to many domains for different resources. Thus, two isolates can share a single resource domain for, say, CPU time, but be bound to distinct domains for outgoing socket traffic.

The RM API does not impose any policy on a domain; policies are explicitly defined by programs. A resource management policy for a resource controls when a computation may gain access to, or *consume*, a unit of that resource. The policy may specify *reservations*[1] and arbitrary *consume actions* that should execute when a request to consume a given quantity of resource is made by an isolate bound to a resource domain. Consume actions defined to execute prior to the consuming act as programmable *constraints* and can influence whether or not the request is granted. Regardless of the outcome, they can arbitrarily modify the behavior of isolates bound

[1]Guaranteed resource availability, which 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.

to the resource domain. Consume actions defined to execute after the consume event can be thought of as *notifications*.

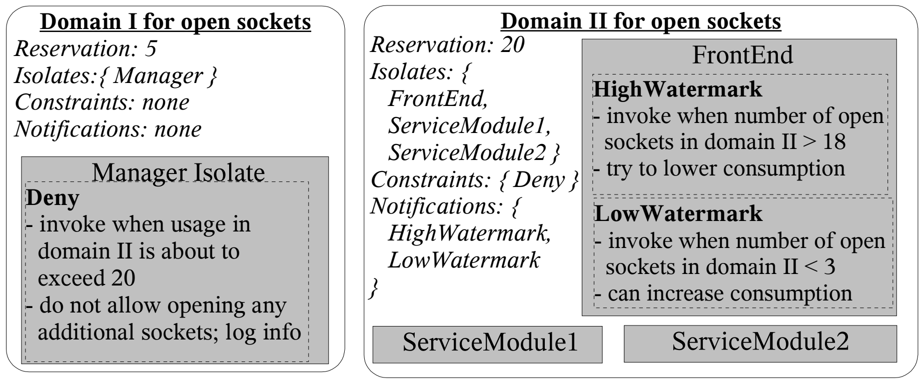
Domain I for open sockets
Reservation: 5
Isolates:{ Manager }
Constraints: none
Notifications: none
Manager Isolate
Deny
- invoke when usage in domain II is about to exceed 20
- do not allow opening any additional sockets; log info
Domain II for open sockets
Reservation: 20
Isolates: { FrontEnd, ServiceModule1, ServiceModule2 }
Constraints: { Deny }
Notifications: { HighWatermark, LowWatermark }
FrontEnd
HighWatermark
- invoke when number of open sockets in domain II > 18
- try to lower consumption
LowWatermark
- invoke when number of open sockets in domain II < 3
- can increase consumption
ServiceModule1
ServiceModule2

Figure 1. The Manager isolate controls three isolates with respect to their consumption of sockets. The controlled isolates are bound to the same resource domain (i.e., they share the same policy). Moreover, one of the isolates created two notifications to be informed about resource shortage or abundance.

Figure 1 illustrates these concepts. In this simplified example there is only one managed resource: the number of open sockets. A *Manager* isolate executes in its own resource domain. It starts three isolates that are bound to a separate resource domain, to ensure that the manager and any other service started later are insulated from consumption by the service provided by these three isolates. The second domain has a reservation of 20. This quantity is guaranteed, but only if there are no constraints further limiting the resource's availability. In our example there is one constraint, *Deny*, which is also set to 20; it logs an appropriate information error message before rejecting the request to open more sockets than allowed. Finally two notifications, *HighWatermark* and *LowWatermark,* are set by the isolates in the second domain themselves to detect when the resource is scarce or plentiful and to modify consumption accordingly.

Resource Attributes The RM API characterizes a resource as a set of attributes. The attributes provide a programmatic means of learning about the properties of a given resource. Resource characterization relevant to this study includes the *disposable* boolean attribute and the numerical *granularity* of management.

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. A file descriptor 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.

The *granularity* of a resource is the minimal indivisible amount of the resource in a given implementation. For example, 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 page size. RM API method arguments that specify resource quantities are automatically rounded up to the nearest granularity multiple. Requesting to consume a non-multiple of granularity is legal, but the returned value will be a granularity multiple. The rationale behind granularity is to allow managing the tradeoff between accounting precision and its cost. This attribute is important for resource implementers; applications are typically insulated from dealing with it directly.

The resource characterization does not provide any attributes related to bandwidth, and correspondingly the RM API provides no direct means of manipulating consumption rates. However, we observe 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. Examples in the following sections expand this point in greater detail.

Resource Dispenser 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 resource domains and thus (indirectly) to isolates. 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 (Fig. 2). 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.

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 dispensers.

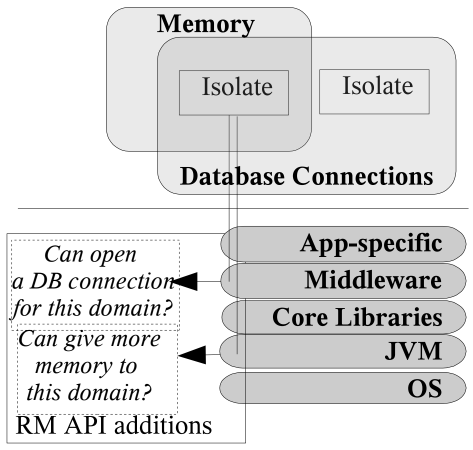

Figure 2. Resource implementations consult their dispensers when granting a request. The decisions depend on policies defined in resource domains.

Defining Resources The task of exposing resources through the RM API belongs to virtual machine implementers, to implementers of the JRE, and to developers of libraries defining resources. To make a resource manageable through the RM API, one must subclass ResourceAttributes and Dispenser, and insert consume() and unconsume() calls where appropriate in the resource's implementation.

Figure 2 illustrates this general approach: computations request resources exactly as they would prior to use of the RM API, and small modifications to resource implementations consult programmable consumption policies, deciding whether a given request can be satisfied.

4 J2EERI on MVM

The J2EE specification defines four application types: Java™ Servlets (JavaServer™ Pages), Enterprise JavaBeans ™(EJB™), Applets and (rich) Application Clients; and an array of services, e.g., Java™ Database Connectivity ("JDBC™") and Java™ Message Service (JMS). Instances of these applications are hosted in containers that interpose between the applications and the set of available services. The containers are themselves hosted in servers, e.g., Web, EJB, and JMS, that may occupy one or more Java virtual machines.

The J2EERI implementation is written entirely in the Java programming language. It includes a version of the Tomcat [11] Web server. The JMS service depends on the Cloudscape relational database [12], which is included with the J2EERI and is also written entirely in the Java programming language. At runtime, all servers execute in the same JVM, including the Cloudscape instance that supports JMS.

In earlier work [8], we evaluated several ways in which isolates might be used to restructure a J2EE server, and the J2EERI in particular. While a straightforward approach would place J2EERI implementation modules in isolates, our preferred strategy, which forms the basis of the work described here, uses *application domain* isolates. Each application domain hosts one (or more) complete J2EE applications, and the associated containers and servers. In addition, the application domain isolates are controlled by a separate administrative isolate. This structure has two main advantages. First, it is easy to manage resources on a per application domain basis since RM is based on isolates. Stated differently, this is an application (service) oriented architecture. Second, communication overheads are minimized. Local, pass-by-reference interfaces may be used, e.g., EJB local interfaces.

Owing to MVM's efficient runtime meta-data sharing, the cost of replicating the containers and services in each domain is small, on the order of 6MB per domain [8]. Parts of the deployment logic and other sharable components are contained in the administrative isolate. In particular, the administrative isolate is responsible for resource domain management and installs the appropriate callbacks during server startup (Sec. 5.2). Communication between the administrative isolate and application isolates uses the isolate link mechanism.

Optionally an application domain may be further subdivided into isolates that host the major subservers, that is, the Web server, EJB server, and JMS servers. We used the single-isolate variant in this research, but note in passing that the two variants are equivalent from a resource management perspective, due to the fact that several isolates may be bound to the same resource domain. Figure 3 shows the isolate structure of a system we have built with an optional load balancer and three application domains.

Load balancer A load balancer directs incoming requests to an appropriate server, based on knowledge of the load on all the servers in a system. Load balancing is standard practice in large J2EE server configurations and is usually realized by a software "plug in" component to a web server. The mechanisms for communication between the load balancer and the servers being balanced are typically ad hoc and server-specific. In principle load balancing can be performed with respect to arbitrary resources, but is typically limited to CPU load and network traffic.

From the perspective of the RM framework, a load balancer is a software component that requries notification of consumption of the appropriate resources on the servers it is attempting to balance. We have implemented a simple CPU load balancer that consists of an HTTP pass-through engine that directs HTTP requests to the least loaded application domain. The load balancing component registers notification consume actions for the CPU resource against all the application domains under its control. It uses the usage information provided in the notification to maintain a load average for each domain. It would be straightforward to add accounting for additional resources into the selection algorithm and we are planning this for future work. A virtue of RM in this context is that the load balancer uses only the standard isolate API and is largely independent of isolate location. In particular, it is possible to load balance to multiple application domains in a single server. This can be used, for example, to direct a specific class of user requests to a lightly loaded (well resourced) application domain, without the need to establish a multi-server infrastructure.

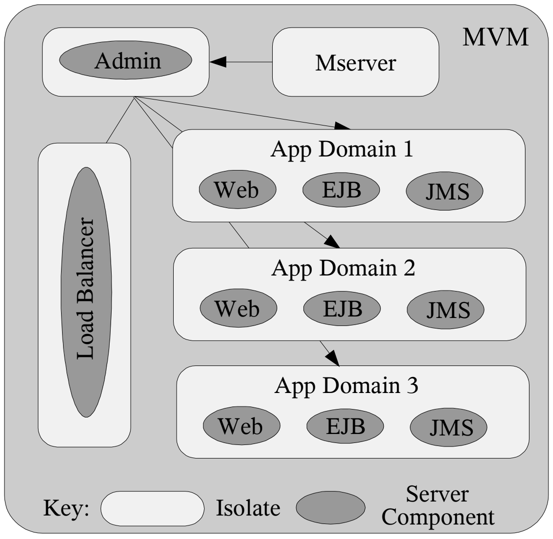

Figure 3: Isolate structure of modified J2EERI.

5 J2EERI Resources

A J2EE server contains many entities that could be characterized as resources in the RM framework. These range from traditional, low-level, resources such as CPU time, memory, network traffic, and threads, to J2EE-specific resources like JDBC connections, EJB container cache, number of active servlets, and so on. One of the main challenges in using RM effectively is the choice of appropriate resources to manage. This is particularly true of the higher level resources, as there are many entities that could potentially be defined as resources.

Two observations can help with choosing appropriate resources:

- The essential requirement of a J2EE server is to deliver a specified level of service at some minimum "cost". The level of service is typically expressed in terms the client of the service can understand, for example, request throughput and latency. Cost might be specified as some function of resources used, typically traditional resources such as CPU time.

- Higher-level resources depend on lower-level resources for their implementation. For example, a JDBC connection requires memory, CPU cycles, and a socket. A servlet may require a JDBC connection. An application may require several servlets and so on.

The first observation suggests identifying the key resources that contribute to the service level and/or have relevance to the cost function. The second observation implies that defining too many resources, particularly if they have complex dependencies between them, is counterproductive. No human administrator could be expected to understand the inter-dependencies sufficiently well to choose appropriate policies. One particular problem with dependent resources is setting conflicting policies that render higher-level resource policies irrelevant. For example, if network bandwidth to a database is set too low, then a policy that attempts to limit transaction rates may be moot if the maxmum rate can never be achieved because of lack of sufficient network bandwidth. The ultimate solution to this problem is coordinated, automated resource management, but that is beyond the scope of this paper.

5.1 J2SE Resources

MVM supports resource management for CPU time, heap memory, sockets, threads, and network traffic, all of which required changes to either MVM itself or to the appropriate core classes. In this research we have explored the management of CPU time in the J2EERI server. This crucial resource illustrates well how to integrate fine-grain management of standard resouces into an application server.

CPU Time Efficient and accurate CPU management is challenging to implement. Unlike other resources, there are no explicit points in the program where CPU resources are requested. If code were always interpreted, the main JVM interpreter loop could be modified to issue consume requests periodically. However, achieving the same effect for compiled code would require consume requests to be embedded in many places in the compiled code, a complicated and expensive solution.

Instead, MVM utilizes a polling thread that periodically wakes up, computes the CPU usage for that isolate in the polling period and then invokes the `consume()` action. This is relatively efficient, depending on the length of the polling period (which, for this resource, is its granularity – see Sec. 3), but does have the property that CPU is actually used before the consume action can impose control.

CPU is an example of a resource that is generally best controlled by limiting the rate of consumption. However, we note in passing that, modulo the polling period, it is straightforward to place an absolute limit on CPU usage to prevent denial of service attacks or to control runaway applications.

The RM API provides a standard rate-adjusting policy that is suitable for many different resources, for example network bandwidth, transaction rate (see Sec. 5.3). The rate is specified as a given threshold value in a given time period, e.g., 100 units in 1 second. The policy is implemented by maintaining a history of resource usage over the interval and, if consumption has been excessive, suspending the isolate on a consume event long enough to bring the rate down to the required level. Given an interval I, a threshold T, history interval H, and usage in history interval U, the formula used to compute the potential length of suspension is $U * (I / T) – H$. If this value is not greater than zero no suspension occurs.

With minor modifications[2], this policy can be applied to controlling CPU usage. For example, 1000ms of CPU time in any 5000ms interval. Or stated in percentage terms, 20% of the CPU in a 5000ms interval.

A variant of this policy simply limits the usage to a specified percentage in every polling period. This is equivalent to the standard policy with the interval set to the polling period and a threshold equal to the given percentage of that period. It has the virtue of requiring no history to be kept, as essentially only the behavior in the most recent polling interval determines the length of suspension, It is also independent of the polling period and arguably more intuitive for an administrator. The virtue of the standard policy is that bursty CPU usage can be tolerated over short periods without suspension, provided that the history interval is large enough. The corollary, however, is that the length of the suspension can be larger in the worst case.

Note that in both variants the potential for overuse is limited to the length of the polling period, which naturally creates pressure to keep it short. However, as we see below, the measurement overhead increases as we reduce the polling period.

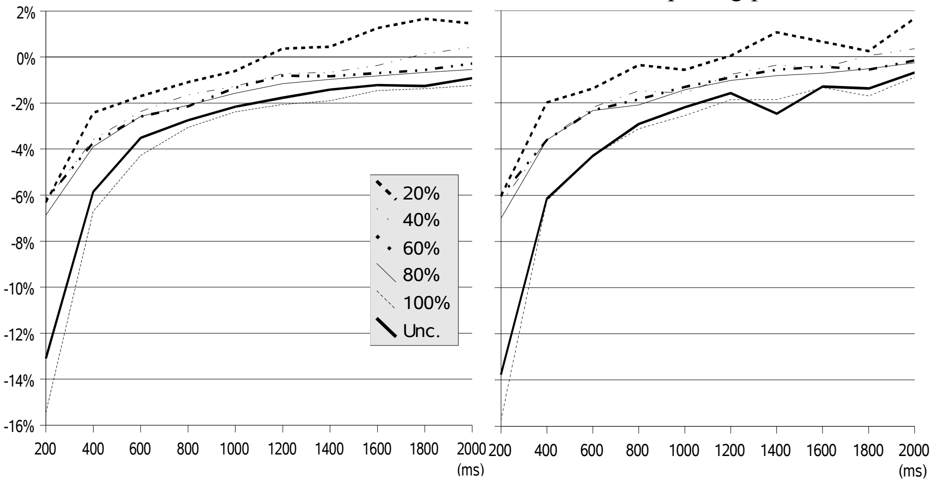

Figure 4. The overhead of CPU time management for the first (left chart) and second (right) variant, as a function of the polling interval, relative to a server with no RM enabled (lower is worse). Each plot corresponds to the CPU percentage allocated to the test isolate.

To measure the overhead of CPU time management we executed a CPU-bound servlet while varying the percentage of allocated CPU and the polling period. Figure 4 shows the percentage reduction in throughput for both variants with polling periods from 200ms to 2s and with varying CPU percentage limits[3]. The baseline for comparison was the throughput of the server with RM disabled, multiplied by an appropriate factor to compute the maximum throughput for different target CPU percentages (0.2 for 20%, 0.4 for 40%, etc.). The measurements show that both

[2]For CPU usage the polling thread handles the consume event, but we must suspend all user threads in the given isolate, and not the polling thread.

[3]All performance data presented in this paper were obtained using the on the Solaris™ 9 Operating System on a Sun 280R server with two 1015Mhz UltraSPARC™ III processors and 4GB of main memory.

methods of controlling allocated CPU time have very similar costs, and at a polling period of 2 seconds, there is no discernible overhead. The *Unc* (unconstrained) data is for a server with RM enabled but no CPU resource callbacks installed and, therefore, shows the overhead of the polling thread. Relative to *Unc*, the 100% case shows the additional overhead of executing the callback. We believe imprecision in the Thread.sleep method to be the cause of the overshoots at low percentage limits and are investigating a more precise implementation.

5.2 Integrating Resource Management into J2EERI

Following standard J2EERI conventions, the set of resources to be managed by the server is specified in a properties file. The file may specifiy a subset of the possible resources under RM control if it is not desired to manage certain resources. The set of possible resources includes those provided by the JRE, e.g., CPU time, and additional resources defined specifically for J2EE. Resources are specified using the standard naming convention of Java classes, e.g., server.resource.JDBCConnection.

The associated classes that RM requires for a resource, e.g., the dispenser and resource attributes (Sec. 3), are assumed to have class names formed by standard suffixes to the resource name, e.g., server.resource.JDBCConnectionDispenser

When MVM starts up, it is passed the file defining the resources to be managed and attempts to instantiate the associated dispenser classes. For reservable resources such as JDBCConnection, the total reservable amount, which is needed by the dispenser, is also specified in the file.

Consume actions applied to a resource are referred to as resource policies and are similarly specified using standardized class names. For example, a consume action that logs the consumption of a resource would be defined and applied in the class server.resource.policy.LogConsumptionPolicy. This class must implement the ResourcePolicy interface and define the method setPolicy.

The application of resource policies is handled by the administrative isolate, again based on information in a properties file. Each application domain enumerates those policies that it wishes to have applied, using the resource class names. The administrative isolate applies a policy by first creating a local resource domain for the resource and binding the application domain isolate to the local resource domain. It then instantiates the policy class and invokes the setPolicy method. For simple policies such as the LogConsumption policy, the setPolicy method simply creates the appropriate consume callback and registers it with the local resource domain. Note that the administrative isolate neither interprets nor depends on the resources or applied policies in any way. Currently the only server code that depends on specific resource types is the load balancer.

Reservations for resources are handled using a generic reservation policy. The amount of the reservation must be specified as an extra argument to the policy in the properties file, and is passed through in a string array to the setPolicy method. Since, initially, the total reservable amount is allocated to the resource domain bound to the administrative isolate, the reservation policy class must first reduce the administrative isolate's allocation and then allocate it to the application domain isolate.

A strength of the RM framework is that generic policies like Reservation and LogConsumption can be specified once and applied to a wide variety of resources.

The following is a simplified version of the LogConsumption policy[4]:

```
public class LogConsumptionPolicy implements ResourcePolicy, Notification {
   public void setPolicy(ResourceDomain admin, ResourceDomain target) {
      localDomain.setPersistentNotification(this);
   }
   public long notify(ResourceDomain d, long previousUsage, long currentUsage) {
      String resourceName = domain.getResourceAttributes().getName();
      long consumedUnits = currentUsage – previousUsage;
      log("Consumed additional " + consumedUnits + " of " + resourceName);
   }
}
```

5.3 J2EE-Specific Resources

From a variety of potential J2EE-specific resources we chose to focus attention on several resources related to database management, namely connections, statements, and transactions. Defining these three resource types allows for controlling both the absolute and rate availability of these resources to application domains (Sec 3.2).

J2EE application performance is often critically dependent on the performance of the underlying database and on the extent to which data can be cached in the application server. We do not address caching mechanisms in this paper but observe that they depend in part on resources such as the size of the EJB caches and the availability of heap memory, that are manageable with the RM framework.

The performance of the underlying database is generally outside the control of MVM/RM as the database system is typically not implemented in the Java programming language. Those databases, such as Cloudscape [12] and Pointbase [13], that are implemented in the Java programming language are rarely used in enterprise system deployments. Nor are the sources readily available, thus making it difficult to apply RM to their implementations.

A database is accessed from a J2EE application using the JDBC API or, more generally, through the Java Connector Architecture (JCA), that provides a standard way to interact with legacy enterprise information systems. The JDBC driver therefore acts a proxy for the database server and several of the JDBC classes effectively represent resoures that are provided by the database. By controlling access to these proxy resources we can, to some extent, control the use of database resources.

In the following sections we describe the chosen resources, their correspondence to database resources, and their integration into J2EERI.

[4]Some of the class and method names in the RM API are different than in the example; the modifications better convey the intuitive meaning behind the code.

Connections

A fundamental JDBC resource is a *connection*, which represents an active session with the database server. A connection is required to perform any interaction with the database. Connections are relatively expensive to set up and a database server can only support a relatively small number of connections, certainly many fewer than the number of concurrent users that might be accessing an application hosted on a J2EE server. A solution to this mismatch is connection pooling, where a fixed number of connections are held open to the database, and shared between the sessions corresponding to the J2EE application clients. An application opens and closes pooled connections in the normal way, but actual database connection open and close operations only take place when necessary, for example, when there are insufficient open connections to handle the load. Connection pool management is one of the more complex areas of a production J2EE server. The administrator typically provides some input to the pool manager in the form of minimum and maximum numbers of open connections. Tuning these values can often have a significant effect on application performance.

Typically, then, from a resource management perspective, a J2EE server can be characterized as having a hard-wired connection management policy, that is tunable by a small set of fixed controls. The hard-wired policy cannot be changed nor is it easy to apply different policies to different databases, beyond altering the fixed controls.

The J2EERI connection pool manager has no externally controllable properties. It attempts to share connections where possible, e.g., when opened with the same transaction, otherwise it grows the pool without limit.[5] It uses a periodic recycling mechanism to detect connections that have not been used recently, closing and removing them from the pool.

To manage connections, we defined a J2EERI server resource called `JDBCConnections`, which is disposable and has a granularity of one (see Sec. 3). It was straightforward to use the `Reservation` policy to limit the maximum number of connections that the pool can open. Each application domain can be assigned a subset of the global limit that is set when the dispenser is created. Further, these reservations can be changed dynamically, although since allocated resources cannot be revoked[6], the reduced reservation only takes effect on a subsequent allocation.

Manually setting fixed connection pool parameters is essentially a trial and error process in current servers and therefore does not adapt well to dynamic changes. While throughput is increased, up to a point, by more connections, database request latency also increases. While an area for future work, it would be possible to use RM to create a connection pool policy that adjusted the number of open connections to maintain optimal throughput, given a target range for request latency.

[5]The underlying database effectively imposes the limit.

[6]Except by destroying the isolate, which releases all of its resources.

Transactions

An entity that is not usually considered a controllable resource in a J2EE server is a *transaction*. Every interaction with a database must take place under a transaction, which ensures the standard ACID properties associated with a database. A transaction corresponds to database server resources, such as data that must be kept in case a transaction needs to abort and locks that are needed to keep concurrent transactions from interfering with each other. Committing a transaction requires action on part of the database server to make relevant data durable, which typically requires disk activity. If the transaction rate is too high the database server can get behind on commits, slowing all clients down.

Controlling the rate of transactions that an application domain can issue is therefore useful. Assuming we can modify the code in the JDBC driver that initiates a transaction to issue a `consume()` action against the transction resource, it is then straightforward to use one of the rate controlling callbacks to limit the transaction rate.

To control transactions we defined a server resource `JDBCTransactions`, which is non-disposable and has a default granularity of 25.

Statements

Transactions typically consist of more than one JDBC statement execution. A JDBC statement is a representation of a SQL statement that is understood by the database server. JDBC provides two essential variants of statement. A plain statement is sent to the database server as a simple text string, requiring the database to compile the SQL before it can be executed. A JDBC statement therefore corresponds to database server resources relating to SQL compilation, which can be resource intensive. The second variant is a prepared statement that the database server compiles once and further can be parameterized to accept different arguments on each call. Prepared statements are therefore less expensive to execute than plain statements.

In certain cases, it might be useful to control statement execution. For example, it would be relatively simple to limit or deny non-prepared statement execution. One could also control the rate of statement execution, although care must be taken not to conflict with controls on transaction execution rate.

To control statements we defined a server resource `JDBCStatements`, which has the same attributes as the `JDBCTransactions` resource.

Experiments

To test the utility and ease of implementation of the above JDBC resources, we modified the MySQL JDBC driver [14] to insert the appropriate RM calls. One issue this immediately raised was that of resource naming. A JDBC driver is required to work with any compliant J2EE server and a J2EE server may utilize several different JDBC drivers. Different servers might wish to define and control different resources and name them differently. Rather than attempt to define a global naming scheme, we instead decided to establish mappings between J2EE server resource names and JDBC driver resource names, that are defined in a server configuration file and propagated to the system properties table. For example, the following mapping is established betwen the server resource name and the driver resource name:

```
server.resource.JDBCConnections= com.mysql.jdbc.resource.connection
```

When the driver wishes to acquire the connection resource domain, it looks up its name for the resource in the system properties and retrieves the server resource name. If the mapping does not exist, the driver simply ignores that resource and does not issue consume calls during execution. Therefore the driver can continue to work with a server that is not enabled for resource management.[7]

The modifications to the driver were straightforward, amounting to a handful of lines in the com.mysql.jdbc.Connection class.

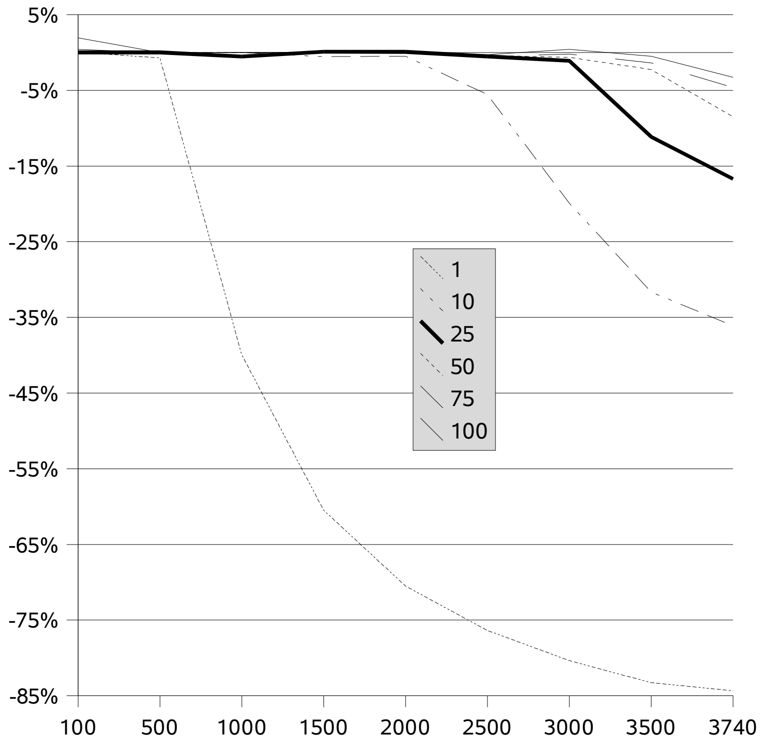

Figure 5. Percentage difference in actual transaction rate against expected transaction rate for different resource granularities.

The rate-based policy mentioned earlier is very suitable for controlling transaction rate. For example to limit an application to 10 transactions per second, one would specificy JDBCTransactions{10, 1000} as the policy.

To test the effectiveness of this policy we used a simple micro-benchmark, similar to that used in the CPU test, that consists of a loop whose body is a transaction that issues an SQL select statement. The benchmark is run for a fixed time interval and reports the transaction rate every 2 seconds. When run with no constraints on transaction rate, the benchmark achieves 3740 transactions per second. To test the

[7]By using reflection the driver can also be independent of the RM API at compile time.

ability to control the rate dynamically, we ran the test with a policy that adjusts the maximum allowed number of transactions from 100 through 3740 in increments of 100, in an interval of 1 second, changing the rate every 30 seconds. To measure the effect of resource granularity, we ran tests with the granularity set to 1, 10, 25, 50, 75 and 100. Figure 5 compares the measured transaction rate against the expected transaction rate. A granularity of 1 adds considerable overhead, which is seen by the sharp reduction at high applied rates. This is to be expected: with granularity equal to 1, the time to consult the current resource policy, which includes an inter-isolate communication, is added to each transaction's execution time. However, at granularities of 50 and above, the maximum reduction is 8.5% and is less than 2.5% across 90% of the range, since only at most 2% of transactions incur the cost of querying the RM framework.

Figure 6 plots the measured and requested transaction rates against time for a subset of the transaction rate range and a granularity of 100, and shows that the measured rate closely tracks the requested rate.

Both of these experiments have very similar results when the controlled resource is the number of statements. This correlation indicates that in some cases these resources are interchangeable with respect to resource management.

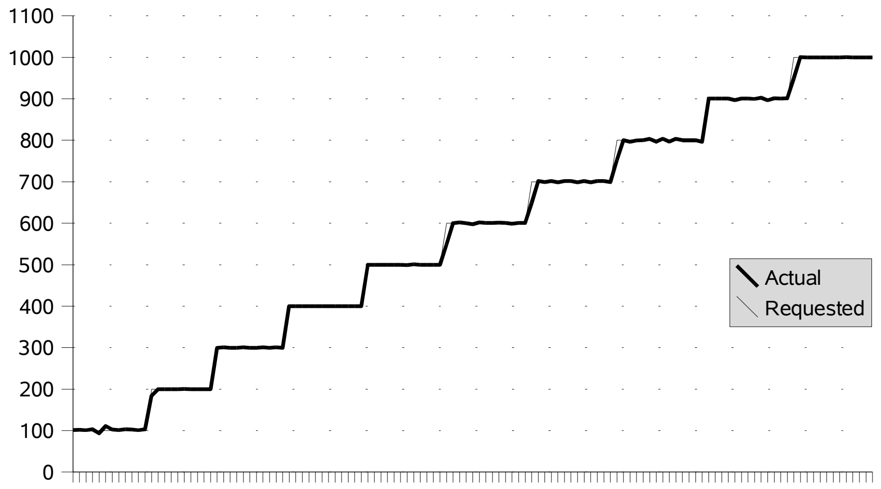

Figure 6. Responsiveness of controlling the rate of transactions.

6 Related Work

J2EE servers currently manage resources they control (pooling of JDBC connections, EJB cache size, number of HTTP server threads) in their own, proprietary ways. Little public information exists on the efficiency of these approaches, and portability problems related to non-standardized configuration management abound. For managing resources not implemented in middleware, application servers fall back on the capabilities of the underlying virtual machine. These rarely go beyond setting heap size or managing the number of threads. However, the underlying operating systems typically provide a richer set of resource management capabilities. Thus, the extent to which a J2EE system can control system resources is dictated largely by its internal partitioning into multiple JVMs.

Falling back on a process abstraction is often inefficient and can suffer from a dual responsibility: in commodity operating systems a process often serves as both a unit of protection and a unit of resource management. Several researchers have pointed out problems stemming from this conflation. For example resource containers [15] were explicitly designed to decouple these two responsibilities. A resource container is associated with a request; the request can traverse multiple protection domains, but all of its resource activities are "charged" against the same container. Resource containers are a generalization of the well-known *paths* abstraction, introduced in the Scout OS [16]. In some respects resource domains resemble resource containers. The resemblance is not complete, though: isolates are required as basic units of resource management in our model, and flexibility is achieved by dynamic association with resource domains.

A resource adaptation framework described in [17] introduces concepts similar to the abstractions existing in the RM API. Most notably the set of resources is extensible and the binding of computations to resources is flexible. The focus of the work is to simplify resource adaptation for middleware platforms through a uniform API and to define interfaces to partition resources among computations, while our work is equally concerned with implementation and performance issues.

We note the existence of management frameworks, such as JMX [3] and JSR-77 [18] that are used in several production J2EE servers. However, when applied to resource management as defined in this paper, these frameworks lack significant functionality. Nevertheless, their high-level approach to management could provide a convenient coordination and provisioning layer, complementing the low-level functionality of the RM API.

The *defensive programming* approach [19] advocates systematic annotation of programs with code watching for denial-of-service attacks. Whenever such abuse takes place, appropriate programmable action is taken. Similarly to that work, we argue the importance of intra-process protection.

In the *staged event-driven architecture* (SEDA) [20], applications consist of a network of stages connected by explicit queues. Each stage admits another request from its queue when this would not lead to exceeding the stage's capacity. SEDA controls each stage's load through mechanisms such as dynamic thread pool sizing and batch size adjusting. Monitoring CPU-related resources (e.g., the number of threads) guides controlling the sizes queue, in effect managing the load of each stage. The architecture presented here can complement SEDA – fine-grain control over a wide set of resource types produces more usage information that can later be fed into admission policy of each of the request handling stages. Coupling SEDA's explicit architectural approach with isolates to create a modular, easy to manage, scalable, overload-averse application server merits investigation.

In [5] we reviewed the state of the art in resource management geared specifically for the Java platform. In summary, systems such as JRes [21] and J-SEAL2 [22] account for CPU time and memory used by programs by a combination of bytecode editing and native code. Neither of these early systems is adequate for our purpose of efficient, flexible, uniform, and extensible management of resources in an application server. Both JRes and J-SEAL2 incur overheads around 20% of the execution time, even if there are no constraints. The set of resources managed is not extensible, and neither system can fully account for resources consumed by the run-time system and some core classes.

Modifications of the virtual machine, such as KaffeOS [23], alleviate some of these problems. KaffeOS implemented the process abstraction within the JVM, so that mistrusting programs can coexist within the same underlying address space. However, KaffeOS's resource management focuses on CPU time and heap memory only and is not designed for application-specific resources.

The Real-Time Java Specification [24], which requires custom virtual machines, is specialized for real-time platforms, with a fixed set of resources, while the RM API's goals include extensibility, flexibility, and cross-scenario applicability that are likely to appeal much more to non-RT programmers.

The .NET platform [25] defines *application domains,* similar to the notion of isolates. Instances of `System.AppDomain` are virtual processes, isolating applications from one another. AppDomains are being used in some .NET servers, e.g., ASP.NET. We are not aware of APIs for managing resources available to individual domains.

7 Future Work

In this research we have only studied a small subset of the total set of resources that could be managed in a J2EE server. We plan to extend the work to additional low-level resources, especially heap memory and threads, as well as to other J2EE resources such as EJB caches.

Work is also underway to extend the Isolate programming model, RM, and MVM to clusters of separate machines. This will allow us to investigate the utility of these mechanisms in large, horizontally scaled, multi-tier J2EE configurations.

However, manual control of resources really does not scale to a system as large and complex as a J2EE server. In that context, the RM framework should be seen as a tool to assist in the automated management of resources that is driven by a service level definition and a resource cost function. Solving this problem in the general case requires a global optimizer that fully understands the resource dependencies and can automate the application of appropriate resource policies. We plan to evaluate the potential for connecting an appropriate optimization engine to the RM framework.

8 Conclusions

An area where safe language platforms seriously lag behind operating systems is that of resource management facilities. In general, multi-resource load balancing and fine-grained and efficient resource usage monitoring are difficult to accomplish 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 solutions and may discourage some developers from using safe languages. The situation, already problematic in J2SE, is exacerbated in application servers, as the deployed applications rely on more services and resources, implemented by J2EE but without standardized and adequate resource management programming interfaces.

In this paper we demonstrated that an application server can be equipped with flexible, extensible, and efficient mechanisms for fine-grained resource management. The presented architecture builds on a virtual machine that supports multi-tasking and is capable of managing standard resources. The prototype enables uniform management of an extended set of resource types, significantly reduces the complexity of provisioning resources for code deployed in application servers, and allows for precise and inexpensive usage monitoring. Moreover, the presented RM extensions and their dependence on isolates as opposed to OS processes can free application server architects and administrators from requiring multiple virtual machines in their designs, further simplifying the design and management of J2EE execution environments.

From the perspective of resource implementers we found two features of the resulting prototype particularly useful: ease of retrofitting existing resource implementations to make them fit into the RM framework, and the programmable cost/precision tradeoff of the RM API.

Even without extensive tuning, the current prototype is already satisfactory for practical use, with very low – in some cases not measurable – overheads. It demonstrates the potential for a complete and controlled environment for enterprise applications and forms a good basis for further research.

References

1. Sun Microsystems, Inc.: Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee/index.jsp
2. Gosling, J., Joy, B., Steele, G. and Bracha, G.: The Java Language Specification. 2nd Edition. Addison-Wesley (2000)
3. Sun Microsystems, Inc.: Java Management Extensions. http://java.sun.com/products/Java-Management/
4. Distributed Management Task Force: Common Information Model (CIM) Standards. http://www.dmtf.org/standards.cim
5. Czajkowski, G., Hahn, S., Skinner, G., and Soper, P., Bryce C.: A Resource Management Interface for the Java Platform. Sun Microsystems TR 2003-124 (2003)
6. Java Community Process: JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp
7. Czajkowski, G., and Daynes, L.: Multitasking without Compromise: A Virtual Machine Evolution. ACM OOPSLA'01, Tampa, FL
8. Jordan, M., Daynes, L., Czajkowski, G., Jarzab, M., Bryce. C.: Scaling J2EE™ Application Servers with the Multi-Tasking Virtual Machine. Sun Microsystems TR 2004-135 (2004)
9. Sun Microsystems, Inc.: Java 2 Platform, Enterprise Edition (J2EE) - Version 1.3.1 Release. http://java.sun.com/j2ee/sdk_1.3/

10. Sun Microsystems, Inc. Java HotSpot™ Technology. http://java.sun.com/products/hotspot
11. The Apache Software Foundation: Apache Tomcat. http://jakarta.apache.org/tomcat
12. IBM Corporation: The Cloudscape database. http://www.ibm.com/software/data/cloud-scape/
13. Pointbase: The Pointbase Database, http://www.pointbase.com
14. MySQL AB: MySQL Connector/J. http://www.mysql.com/products/connector-j/
15. Bang, G., Druschel, P., and Mogul, J.: Better Operating System Features for Faster Network Servers. Wokshop on Internet Server Performance, Madison, WI (1998)
16. Mosberger, D., and Peterson, L.: Making Paths Explicit in the Scout Operating System, Proceedings of OSDI '96, Seattle, WA (1996)
17. Parlavantas, N., Blair, G.S., and Coulson, G.: A Resource Adaption Framework for Reflective Middleware. Proceedings of the 2nd Intl. Workshop on Reflective and Adaptive Middleware, Ri Janeiro, Brazil (2003)
18. Java Community Process: JSR-77: Java 2 Enterprise Edition Management Specification. http://jcp.org/jsr/detail/77.jsp
19. Qie, X., Pang, R., and Peterson, L.: Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. 5th OSDI, Boston, MA (2002)
20. Welsh, M., Culler, D., and Brewer D.: SEDA An Architecture for Well-Conditioned, Scalable, Internet. 18th SOSP, Banff, Canada (2001)
21. Czajkowski, G., and von Eicken, T.: JRes: A Resource Accounting Interface for Java.
22. Binder, W., Hulaas, J., and Villazon, A.: Portable Resource Control in Java: The J-SEAL2 Approach. 16th ACM OOPSLA, Tampa Bay, FL (2001)
23. Back, G., Hsieh, W., and Leprau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th OSDI, San Diego, CA (2000)
24. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Hardin, D., and Turnbull, M.: The Real-Time Specification for Java. Addison-Wesley (2000)
25. Microsoft Corp. .NET Web Page. http://www.microsoft.com/net (2002)