

Adaptive Middleware for Dynamic Component-Level Deployment

Mick Jordan

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025
Mick.Jordan@sun.com

Christopher Stewart

University of Rochester
734 Computer Studies Bldg
Rochester, NY 14627
Stewart@cs.rochester.edu

ABSTRACT

Utility hosting centers can reduce resource consumption by optimally distributing and load balancing their client's application components. However, current state-of-the-art middleware systems either rely on the application to deploy itself, and/or provide the hosting center with coarse-grained and static deployment mechanisms. In this paper, we present our adaptive middleware system and API that allows the hosting-center to dynamically exchange component-level deployment generators for all hosted applications. To demonstrate our system's capabilities, we provide experimental results on the speed and effectiveness of dynamic deployment strategies.

Categories and Subject Descriptors

D.4.7 [OPERATING SYSTEMS]: Organization and Design;
D.2.11 [SOFTWARE ENGINEERING]: Software Architectures;

1 INTRODUCTION

In the utility computing business model, a utility computing center (UCC) provides QoS guarantees for client application providers in an on-demand fashion. For a UCC, the energy and infrastructure costs of its cluster depend on the amount of cluster resources required to satisfy client demands. To maximize profit, a UCC should carefully deploy, replicate and load balance an application for minimal resource consumption consistent with meeting the QoS requirements. Essential to any effective deployment strategy is the ability to isolate applications and control their resource usage. Traditional operating systems (OS) provide strong support for address-space isolation, but weak or rigid support for resource isolation.

There are many reasons behind the drive for component-based applications, ease of replication, recovery, reuse, etc., although there is as yet no agreement on exactly what form a component should take. However, strong isolation would seem to be an important characteristic in the UCC context.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RM'05, November 28-December 2, 2005 Grenoble, France.

Copyright 2005 ACM 1-59593-270-4/05/11...\$5.00

Component-based applications place higher demands on intelligent deployment strategies but have the potential for greater pay-off. Using component-level replication and load-balancing, deployments can be tailored to the resource requirements of individual components, which is more fine-grained than the blanket assumptions made using whole-application deployments. For instance, a CPU intensive component can be replicated on machines with dedicated processing power, without fear of other components consuming precious CPU cycles.

The conflation of the traditional operating system (OS) process abstraction and the unit of submission to a typical UCC facility makes component-level deployments difficult in existing environments. The UCC system controls the placement of individual processes and typically does not provide support for custom strategies for jobs consisting of multiple components. Further, the highly OS-dependent application programming interface (API) to the process mechanism and the widespread use of scripting languages over general-purpose programming languages, works against the development of portable, flexible and extensible deployment strategies.

In this paper, we invoke the principle of separation of concerns to propose a robust middleware system that supports low-level deployment mechanisms with adaptive support for plug-n-play deployment generators. We believe that middleware developers should be concerned with creating a simple interface to underlying deployment mechanisms such as load balancing and component replication. The UCC administrators should focus on the development of high level deployment policies, without regard for the underlying implementation. However, even with these facilities, application developers must still provide some meta-data about the interfaces to their components to support dynamic placement.

The focus of our system is in supporting long-lived, scalable enterprise services, such as those typically accessed over the internet, rather than high-performance technical computing applications. A service-hosting UCC can be seen as a special kind of Grid system [6], that provides application deployment capabilities to service developers and client access to the deployed services.

The remainder of this paper is as follows. Section two describes related work. This is followed in Section three by background on the platform on which our system is built, Section four describes the system design and this is followed by sections on experimental results, future work and conclusions.

2 RELATED WORK

Current utility hosting systems, like the Service On-Demand Architecture (SODA) [10], create and manage virtual containers. Virtual containers consume a portion of a single cluster node's resources to run a client service, and can be provisioned on-demand. Application replication is achieved by executing client services on multiple virtual containers. Our work is inspired by and closely related to SODA, but differs in that we seek a portable solution based on the Java™ platform, and pluggable deployment generators.

In a traditional computing environment applications are manually deployed, and may include application-provided code that customizes the deployment. Standard tools are available to ease the process [7], although these may not be adequate for more complex applications such as typical enterprise systems. Furthermore, deployment (installation) is typically considered to be a one-time activity and little support is provided for dynamically installing or re-configuring a deployed system. This is inadequate for the UCC environment where applications may need to be deployed on different machines and in different configurations on a regular basis in response to changing demands.

Several component-level deployment techniques have been proposed [16, 5, 13]. These techniques range from model-based profiling to adaptive control, and are not easy to implement. A goal of our work is to provide a realistic but easy to framework that supports the development of such techniques.

3 UNDERLYING INFRASTRUCTURE

Our system is built on an experimental Java platform (MVM) [2] that supports multiple concurrent applications (isolates) and provides for the flexible management of resources.

3.1 Isolates

MVM implements the Application Isolation API (Isolate API), which is defined by JSR-121 [9]. The Isolate API virtualizes a Java Virtual Machine™ (JVM™), allowing new isolates to be created and terminated programmatically.

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 modification.

Starting a new isolated computation is similar to creating a new thread and amounts to specifying the main class and arguments for the new isolate and invoking its `start` method:

```
Isolate i = new Isolate("Hello", new String[] {});  
i.start();
```

The Isolate API lends itself to different implementation strategies. The implementation strategy employed by MVM is for all isolated applications to reside within a single JVM instance that implements the protection boundaries within a single address space.

MVM instances on multiple compute nodes may also be managed as a cluster. We use the term *aggregate* to describe the entity that manages the collection of isolates associated with a single MVM instance. A simple extension of the Isolate API provides for transparent or controlled placement of isolates in aggregates.

Isolates provide a convenient container for either traditional monolithic applications or applications structured as isolated components. Isolates provide a more portable and easier to use API than operating system processes, and a collection of isolates

are capable of a more efficient implementation than the equivalent set of JVM processes. The principal disadvantage is that if the MVM fails, all isolates fail, but this is the same for operating systems and processes. Evidently MVM needs to be engineered to match the level of robustness and reliability of an operating system.

3.2 Resource Management

MVM also provides resource management capabilities through the implementation of a flexible API (RM). The dependency between the RM API and the Isolate API is that the unit of management for the RM API is defined as an isolate. This choice makes accountability unambiguous, as each resource in use has exactly one owner.¹

This section contains a brief overview of the main features of the RM API [3]. It is important to note that all existing Java applications can run unmodified under RM, even if the classes they depend on exploit the RM system. Applications and middleware that need to control how resources are partitioned (e.g., utility computing services) can use the API for that purpose. The API can also be used 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.

Key abstractions of the RM API are discussed below.

Resource Domain: A *resource domain* encapsulates a set of usage *policies* for a resource. All isolates *bound* to a given resource domain are uniformly subject to that domain's policies for the domain's 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 JDBC™ connections.

Constraints and Notifications: 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* 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 may be defined to execute prior to the consume request and serve as programmable *constraints* that can influence whether or not the request is granted. Consume actions may also be defined to execute after the consume event and serve as *notifications* of resource consumption.

Defining Resources: Resources are exposed through the 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 take advantage of the RM API is relatively easy.

A particular resource is represented by an instance of a Java class that must inherit from the `ResourceAttributes` class, part of the RM API, that provides methods that define standard resource characteristics:

¹ In [3] we discuss problems associated with designating class loaders or threads/thread groups as principals in resource management schemes.

- **Disposable:** A resource is disposable if it can be returned and reused at a later time. E.g., JDBC Connections are disposable but CPU cycles are not.
- **Unbounded:** there is no fixed limit on the amount of resource available. CPU cycles are an example of an unbounded resource and Memory is an example of a bounded resource.
- **Reservable:** Bounded resources may be reserved by a domain for future consumption, up to a pre-determined limit.

Dispensers: Dispensers model the notion of a point of manufacture for a resource. For example, a storage management system "manufactures" memory. Dispensers record the total available quantity of a resource, manage resource reservations and coordinate the consume and unconsume actions associated with resource domains.

Resource classes and their associated dispensers are typically loaded dynamically using the dynamic loading mechanisms of the Java platform, thus systems may run with different sets of managed resources.

4 SYSTEM DESIGN

4.1 Service Containers

Application components are executed in *service containers*. A service container is similar to mechanisms provided by some operating systems, e.g. Resource Containers [1] Solaris™ Zones [18] or SODA Virtual Service Nodes [10], but is implemented entirely at the Java level using isolates and the RM framework.

A service container may host one or more isolates. A service container has an associated set of RM policies that apply to all the isolates hosted in the service container. Unlike the similar OS-based entities, a service container has no pre-ordained resource partitioning. For example, a set of service containers could be created that were partitioned based on a high-level resource such as the number of JDBC transactions per unit time. However, the default partitioning is in terms of the traditional concrete resources, namely CPU cycles, memory and network bandwidth. In addition to changing the resource allocations of a service container, e.g., the share of CPU time, it is also possible to change the set of policies associated with a service container dynamically at runtime.

Service containers may be arranged in a hierarchy, with the rule that only leaf service containers may host isolates. Leaf service containers may not span MVM aggregates, i.e., the isolates they host all reside on the same aggregate. Typically there is one aggregate per physical machine. However, non-leaf service containers may span aggregates and therefore act as containers for multi-component services that are deployed on multiple aggregates.

Service container creation and deletion is a very fast operation. Since MVM shares the majority of meta-data, e.g., class bytecodes, between multiple instances of the same class, creating a new service component in a service container typically is also a fast operation.

The service container subsystem is a self-contained layer, depending only on isolates and RM. It provides an API to create destroy, and manipulate service containers, install and run isolates and provide status information.

4.2 Service Deployment

Service containers provide the basic mechanism for resource control and application (service) component installation. In order for a UCC to efficiently utilize the resources under its control it must deploy service components to appropriately sized service containers, and possibly expand or contract the number of service containers.

We assume that the majority of application components that are hosted on the UCC are either freely replicable, e.g. stateless web servers, or are cluster aware in the case of stateful services such as data store components. The basic deployment mechanisms provide for components to learn about the location of their peers if required.

4.3 Component-Level Replication

While the service container approach is mostly transparent for application-level replication, component-level replication is more difficult. Consider, for example, a J2EE™ [17] application that consists of a set of servlets, each of which implements one aspect of the service. To be concrete we shall describe the RUBiS synthetic benchmark [15] that we use in our experiments. RUBiS simulates an auction internet site, allowing browsing of the available items and managing auction bids. It consists of fourteen separate servlets, which are stateless, and therefore freely replicable, since the data is stored in a relational database.

In practice, the distribution of user requests is not uniform across the set of servlets. For example, browsing is more common than bidding. Furthermore, a service-level agreement for an auction site may well require that bidding requests are processed faster than browsing requests. To achieve this differentiation of service will require that the individual servlets be deployed differently in the UCC. Unfortunately servlets are not a component in the typical UCC sense, indeed their deployment is handled by an application server that is just another application as far as the UCC is concerned. To finesse this, each servlet would have to be deployed individually in a separate instance of the application server in order to fit the UCC application deployment model. This would add tedious administrative overhead to the development process as each servlet would have to be placed in its own war archive and separate application server configurations would have to be created, one per servlet.

The isolate-based service container approach does not provide a direct solution to this problem, since a servlet is not an isolate.² However, servlets and similar components would undoubtedly benefit from improved isolation, and this could happen were isolates to become part of the standard Java platform. Our current solution to specify the component (servlet) as *embedded* in the application server component. This allows the deployment system to treat the component as independent for the purposes of deployment analysis but actually instantiate an application server instance with the deployed servlet(s) in a service container. As described in more detail in section 4.5, we achieve the effect of component-level deployment through the load balancer, leveraging the dynamic class loading mechanisms of the Java platform.

² See [12] for a discussion on the difficulties of achieving this in current application servers.

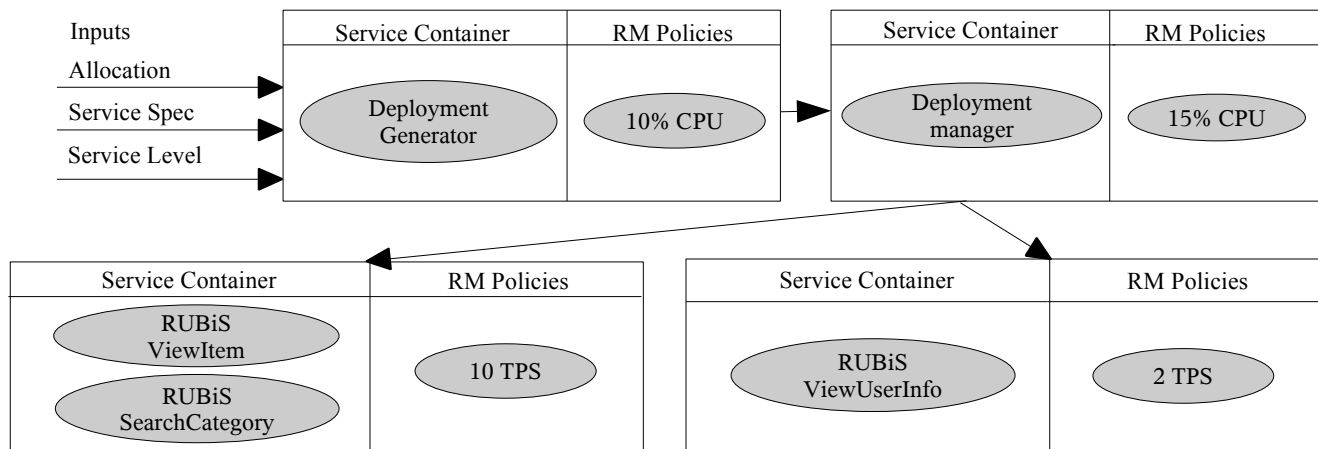


Figure 1. Overall system architecture.

4.4 Replication Support

Typical enterprise applications (e.g. application servers) maintain considerable state in the file system, and this must be replicated along with the code of the application. Our middleware manages the replication of this file system data with only minimal help from the application itself. The application provides the system with the path to the root of a directory tree that is cloned before a new replica is started. Since the file system typically contains machine-specific configuration information in a format known only to the application, the application must also provide a configuration class that is dynamically loaded and invoked after the cloning is complete.

For most configurable applications, the configuration class is straightforward to write. For instance in a typical application server, it simply modifies a few configuration files.

4.5 Deployment API

Deployment generators are programs that make deployment decisions for an application. Historically they have tended to embody both the mechanism and the policies required for deployment to take place. Because deployment mechanisms are often application-specific, deployment generators have historically been non-portable. To avoid this we separate the deployment mechanisms from the policies that invoke them, which we can achieve by leveraging service containers as an application-agnostic unit of deployment.

An optimal deployment generator must accurately predict application behavior. Deployment generators can range from simple to sophisticated, and application-agnostic to application-specific. For example, off-line profiling of an application might produce a fixed deployment strategy that is read from a file on startup and does not change once a deployment is complete.

However workload surges, changes in service level demands, and application performance anomalies make static deployments inadequate in the UCC context. As a result, deployment generators may be highly parameterized, and require constant tuning for near-optimal performance.

We believe that a UCC should support more than one deployment generator and also allow dynamic exchange of deployment generators. By deploying deployment generators themselves in service containers, we ensure their isolation from

the deployed applications, and can both limit and guarantee their resource consumption and also change them dynamically.

The overall architecture is illustrated in figure 1, which shows part of an example RUBiS deployment. The maximum transaction rates of the two service containers containing the RUBiS components are different, reflecting the different request frequencies expected for those operations. The deployment generator and deployment manager are both provided with minimum CPU guarantees.

Deployment generators are provided with a service container allocation, service specifications, service level requirements, and can also access service performance data. Using this information, the generator can generate new deployments and submit them to the UCC deployment manager. The output of a deployment generator is essentially a map from components to service containers. The deployment manager, which is common to all generators, simply compares the new deployment to the existing deployment (if any) and mechanically adds or removes service containers as necessary to meet the requirements of the new deployment. I.e., all the deployment strategy logic is contained in the deployment generator.

4.6 Load Balancing

The middleware includes an extensible HTTP load balancer that supports component-level replication, by allowing a service-specific class to choose targets for incoming requests. A simple round-robin scheme is the default implementation of this class. However, a service may provide its own load balancer since these are treated as (special) components by the deployment system.

It is important that the deployment system integrates with the load balancer, so as to keep it informed of changes in the deployment configuration. To achieve this, load balancer components are specially designated in the service description file and must implement an extension of the normal component configuration interface. Whenever a component is added or removed from a service deployment, the load balancer configuration class is invoked with this information, which can then be passed on in some way to the active load balancer. Typically a deployment generator will provide a load-balancer component with a dedicated service container, but this is not a requirement.

We wrote a custom target generator to support the RUBiS service. Using the mechanisms described in the previous paragraph, the target generator receives information on the deployment of the servlet components any time that the deployment changes. Each deployment actually includes the application server and the complete set of RUBiS servlets. However, since the servlet code is dynamically loaded by the application server on demand, we can achieve the effect of a servlet-level deployment strategy by only sending requests for particular servlets to the service containers that have been designated to host those particular servlets. The load balancer and the deployment strategy generator therefore work together to achieve effective load balancing. The deployment generator makes the basic decisions on which servlets are placed in which service containers and which are replicated. Given that information, the load balancer selects the set of possible servers for a given request and, in the case of replicated instances, attempts to choose the optimal server given the current load.

4.7 Experimental Results

We provide some initial experimental results as a proof-of-concept.

The version of MVM that we used is based on the public release [14], which is compatible with Java™ 5.0, extended with support for resource management and clustering. The extended version is expected to be released in the near future. This version of MVM has only recently become stable, and the clustering and resource management extensions are still a work in progress, which has limited our ability to provide detailed experimental results.

The deployed application is RUBiS hosted on the JBoss 4.0.2 application server [8]. We used a simple custom workload generator to drive user workloads and measure throughput and response times. The workload is based on a file of about 2000 randomly generated requests that was generated with a tool that is provided with the RUBiS distribution. The particular set of requests that we used are biased towards browsing activities. The load generator uses multiple threads to send concurrent requests chosen at random from the file. To stress the server, requests are sent continuously with no inter-request delay.

To relate the performance of the service containers to the RUBiS application, we created a new abstract resource to represent a RUBiS transaction. This is similar to the existing resource for generic JDBC transactions [11] but hides exactly how many JDBC transactions are used in each RUBiS transaction (it is not 1-1), and allows the system to be controlled from the perspective of a client of the system. I.e., a service level agreement would typically be specified in terms of application-level transactions (requests).

Adding this new resource required a simple modification to RUBiS to install the appropriate consume call in the servlet's execution path. We then used one of our standard rate-limiting policies to create service containers capable of delivering a fixed number of RUBiS transactions per second – we used a value of ten for the experiments. The resulting effect is that the underlying physical computing infrastructure is virtualized into service containers that are capable of delivering application-specific service levels, in this case ten RUBiS requests per second.

To test the system we implemented a simple deployment generator that was provided periodically with a fixed allocation of the RUBiS-specific service containers. The number of service containers increased by one at each period, simulating, for

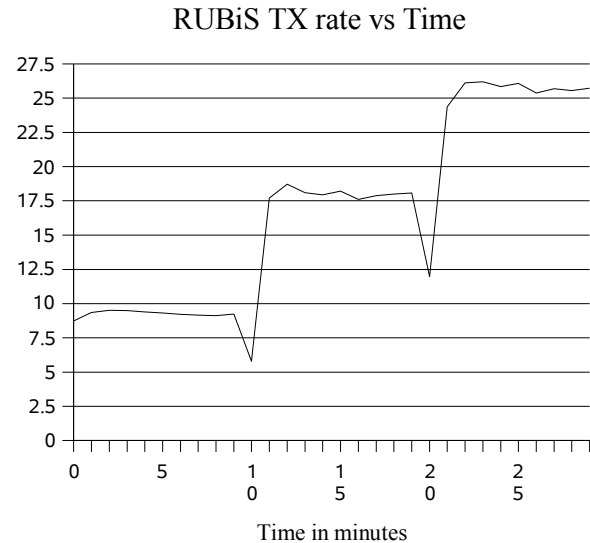


Figure 2: Adding servers over time

example, a manual increase in allocation by a system administrator in response to increasing load. The simple deployment generator simply allocates all components (RUBiS servlets) in a fresh JBoss instance to each container. The average time taken to effect the service-container specific replication was 3 seconds, which includes copying the JBoss file-system tree and configuring the instance with unique port numbers for its services. JBoss itself takes between 45 and 60 seconds to boot,³ hence the contribution from the replication is negligible.

Figure 2 corresponds to a run where a new container is added every ten minutes, and shows the growth in transaction rate as new containers are added. It is important to note that the only control input to the system is the notification to the deployment generator of an increase in service container allocation, after which the system adapts automatically by creating the new container, installing, configuring and starting the server. Owing to some performance problems when using the clustered MVM implementation at the time of writing, this test was run on a single Sun Netra™ SMP, with 12 processors and 96GB of memory. The aggregate transaction rate at each level is somewhat below that expected, as each additional container delivers slightly less than the expected 10 tx/s. We believe this to be caused by overhead in the MVM/RM implementation, that we are investigating. Note also that the service containers do not have any policies controlling CPU usage, which explains the drop in throughput when new server instances start. Server startup is CPU intensive and steals cycles from the running service containers. In the future this will be remedied by adding CPU resource control to the service containers, guaranteeing them a level that is sufficient to meet the desired transaction rate. The trade-off is that, on a single machine, the time for a new server instance to startup would increase.

³ This version of MVM does not share classes loaded in user-defined classloaders, which are used extensively by JBoss, so startup time does not reduce for additional instances as is normally the case with MVM [12].

Although not shown in the results, it should be evident that the system would adapt by shutting down servers in response to a reduction in container allocation. Similarly, a more sophisticated deployment generator that monitored the throughput of the system,⁴ would adapt to both failures to meet services levels, or changes in service levels by adjusting the allocation of components. A service-specific deployment generator, for example, one that monitored the frequency of particular requests to specific RUBiS servlets, would adjust the allocation of servlets to containers appropriately.

5 FUTURE WORK

The main area of future work will focus on the development of more sophisticated deployment generators, for example, based on emerging concepts such as reinforcement learning [4], [19].

Currently we are using ad hoc XML-based techniques for specifying components and services, service level agreements and service container allocations. We expect to move towards emerging industry standards in these areas.

We also plan to integrate the system with simple generic Grid system, also based on MVM, so that services can be installed and monitored through its standard external mechanisms.

6 CONCLUSIONS

Increasingly, applications are being constructed using components and deployed in utility computing environments. Research into suitable deployment strategies for component-based systems is advantaged by flexible and adaptive middleware that simplifies the deployment process and allows experimentation with different deployment strategies.

We have described a system that builds on the Multi-tasking Virtual Machine and its associated Resource Management framework systems to provide the notion of a service container that can partition the resources available to an application component. The system makes extensive use of the reflective and dynamic capabilities of the Java platform to allow new services and new deployment generators to be added to a running system. Basic deployment mechanisms, with the opportunity for application specific customization, are provided on top of service containers that can be driven using a simple API from a variety of different deployment generators. A customizable HTTP load balancer is provided that can be used to gain the effect of fine-grain component-level deployment, e.g., servlets. The system is capable of supporting replicated instances of large applications such as application servers, with very fast deployment of new instances.

The resulting system provides a convenient and practical platform for research into deployment strategies for utility computing environments.

7 ACKNOWLEDGEMENTS

The authors wish to thank the anonymous referees for their constructive feedback.

8 REFERENCES

[1] Banga, G., Druschel, P. and Mogul, J. Resource Containers: *A New Facility for Resource Management in Server*

⁴ Throughput and response times are available from the load balancer.

- Systems, Proc. 3rd Symposium on Operating System Design and Implementation*, New Orleans, 1999.
- [2] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution*. 17th ACM OOPSLA'01, Tampa, FL, October 2001.
- [3] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. *A Resource Management Interface for the Java™ Platform*. Sun Microsystems Laboratories Technical Report 2003-124. June 2003.
- [4] Dowling, D. and Cahill, V. *Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning*, Proc. WOSS'04, October 2004, Newport Beach, CA.
- [5] Doyle, R., Chase, J. et al. *Model-Based Resource Provisioning in a Web Service Utility*, Proc. 4th Symposium on Internet Technologies and Systems, Seattle, WA, March 2003.
- [6] Foster, I., *The GRID: Computing without bounds*. Scientific American. Vol 228, Issue 4.
- [7] <http://www.installshield.com>.
- [8] <http://www.jboss.org>
- [9] Java Community Process. *JSR 121: Application Isolation API*.
<http://jcp.org/jsr/detail/121.jsp>.
- [10] Jiang, X. and Xu, D. *SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms*, Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, WA, June 2003
- [11] Jordan, M., Czajkowski, G., Kouklinski, K., and Skinner, G. *Extending a J2EE Server with Dynamic and Flexible Resource Management*. ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, ON, October 2004.
- [12] 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)
- [13] Liu, X., Zhi, X., Singhal, S. and Arlitt, M. *Adaptive Entitlement Control of Resource Containers on Shared Servers*, HP Laboratories, October 2004, HPL-2004-178.
- [14] The MVM Project, <http://mvm.dev.java.net>.
- [15] Rice University Bidding System. <http://rubis.objectweb.org>
- [16] Shen, K., Tang, H., Yang, T. and Chu, L. *Integrated Resource Management for Cluster-based Internet Services*, Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (Boston, MA, December 2002).
- [17] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE)*. <http://java.sun.com/j2ee/index.jsp>
- [18] Sun Microsystems, Inc. *Solaris Zones*, <http://www.sun.com/bigadmin/zones>.
- [19] *Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems*. Sun Microsystems Laboratories, Report Number: TR-2005-141, Feb 1, 2005