

Resource Management for Clusters of Virtual Machines

Grzegorz Czajkowski, Michal Wegiel, Laurent Daynes, Krzysztof Palacz,
Mick Jordan, Glenn Skinner, Ciaran Bryce[‡]

Sun Microsystems Laboratories
Menlo Park, CA, USA
{firstname.lastname}@sun.com

[‡]University of Geneva
Geneva, Switzerland
bryce@cui.unige.ch

Abstract

Enterprise applications are increasingly being built using type-safe programming platforms and deployed over horizontally scalable systems. Horizontal scalability depends crucially on the ability to monitor resource usage and to define and enforce resource management policies capable of guaranteeing a desired service level. However, current safe language platforms have very limited support for resource management, and their cluster-enabled versions reflect this deficiency.

We describe an architecture of federated JavaTM Virtual Machines. Its distinguishing feature is an integrated resource management interface that addresses the above issues. It offers programmatic control over monitoring and controlling the allocation of resources to applications and their components. The scope of each policy can span multiple nodes, realizing fine-grained control. New resource types can be defined and integrated into the framework. Remote management of local resources and the notion of cluster-global resources form a powerful combination capable of expressing policies that achieve effective performance isolation for cluster applications.

1. Introduction

Cluster computing aims at capturing the promise of horizontal scalability offered by interconnected computers. Safe languages, in conjunction with their associated component architectures, such as J2EETM and .NET's managed components, address the problem of the development of portable, multi-component applications. Numerous designs [17, 1, 2, 4, 9, 15, 16, 18] of the Java Virtual Machine (JVMTM) enhanced for cluster computing reflect the importance of both safe language platforms and clusters of computers. However, all of the systems we are aware of have at least one of the following shortcomings: (i) lack of a well-defined container mechanism with guarantees strong enough to enable effective performance

isolation for cluster applications, (ii) only coarse-grained resource management (e.g., in the case of CPU time, granularities often are the number of processors, computers, or whole-machine load), (iii) only mono-resource management (typically only of CPU time), and/or (iv) ability to monitor only, without control-exercising capabilities.

To a large extent, these deficiencies follow from weak support for resource management, especially for defining and enforcing resource consumption policies, in the Java platform. However, they are also present in many cluster programming environments not based on safe languages [10]. Coarse granularity may be sufficient in specialized settings (e.g., long-running, CPU-intensive tasks), but leads to inefficiencies when considered in conjunction with another trend: toward applications consisting of a dynamic population of components that vary in their resource needs and lifetimes. Resource management oriented towards controlling processor usage is applicable to CPU-bound applications, but is inadequate in situations where using network, databases, or memory dominates. Overall, current cluster programming systems based on safe languages offer scant information on resource consumption and only rudimentary mechanisms for controlling a small set of resources. These limitations constrain what can be done with respect to load-balancing and service provisioning.

In this paper we describe an architecture of federated virtual machines acting in concert to execute applications on networked computers. Its distinguishing feature is a flexible, extensible, and efficient resource management API, that we view as essential to any cluster-wide operating environment.

The design follows a trajectory that began with the introduction of the abstraction of an isolated computation, (dubbed *isolate*) [12]. Isolates do not share state among one another and thus each consumed resource has precisely one owner. Owing to this property, isolates can be terminated asynchronously and their resources cleanly reclaimed. Such unambiguous accountability paved the way for the Resource Man-

agement API (RM API) [8], which allows for programming resource consumption policies and binding groups of isolates to them. Policy mechanisms include reservations, constraints, and notifications. The API is extensible: new resource types can be defined to better reflect the requirements of a given application. Although in many cases “traditional” resources such as CPU time, heap memory, or amount of data transferred over the network sufficiently characterize applications’ requirements, there are cases where analyzing different resources offers more insight and abstracts away irrelevant details. For example, we found that controlling the number of transactions can be more useful than direct management of underlying lower-level resources [13].

The contribution of this paper is an architecture for cluster computing in the Java platform. Its distinguishing feature is an integrated and comprehensive support for resource management. Resource consumption of tasks whose components span multiple computers can be monitored and controlled from any node. Performance of applications can be isolated from one another and within an application its components can execute under different policies. The architecture is backward-compatible (existing code runs unmodified).

The infrastructure is well-suited for the execution of multi-component enterprise applications over cluster architectures. Rather than statically dedicating a subset of the nodes of the cluster to each of the applications, it is possible to express multi-application, cluster-wide policies and enforce them via the RM API. A simple example is a policy that manages five applications on three nodes in such a way that, regardless of how many components (if any) of an application execute on a given node, the application gets its equal share of the CPU and network resources of that node, and each application gets at most 1/5 of main memory available on all nodes.

The next two sections introduce the APIs. Examples of policies applicable to applications spanning multiple nodes appear next. Selected aspects of our prototype implementation, based on the Multi-Tasking Virtual Machine (or MVM) [7] follow, along with an analysis of several experiments. An overview of related work and a summary conclude the paper.

2. Background

This section introduces the key abstractions of the original, single-node versions of the Isolation and Resource Management APIs. Most cluster programs will need only these, as they naturally apply to the cluster case, and the platform takes care of placement

of new isolates (Sec. 5). Code such as load-balancers, application managers, or applications that have very specific needs may use cluster extensions of these APIs directly, as described in the next section.

2.1 Isolates

An *isolate* is a container for executing an arbitrary application written for the Java platform. It offers the same guarantees of execution as those provided by executing an application with a JVM. In particular, an isolate does not share any objects with other isolates. Isolates communicate via data-copying mechanisms (sockets, RMI, etc.). Isolate creation and life cycle management are the subject of the Application Isolation API (“Isolate API”), the formal output of JSR-121 [12]. The Isolate API is fully compatible with existing applications and middleware. In particular, applications that pre-date JSR-121 may be managed by the API without modification. We have developed a research variant of the Isolate API; once the JSR 121 is finalized we are planning to adopt the standard.

Programming with isolates bears similarities to programming with threads: launching a new isolated computation amounts to specifying the main class and arguments and invoking the `start()` method:

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

The Isolate API lends itself to various implementation strategies. The one employed by our platform, MVM, is to execute all isolates in a single address space using a JVM equipped with mechanisms for enforcing protection boundaries.

2.2 The Resource Management API

Existing code can run without modification under the the RM API. 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 to them (response time, throughput, footprint, etc.) or to ward off denial of service attacks.

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

The key abstraction of the RM API is a *resource domain*, which 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

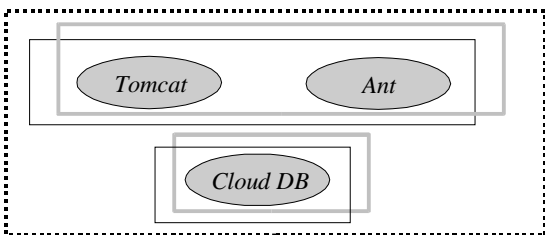


Figure 1. Isolates bound to resource domains.

more than one domain for the same resource, but can be bound to different 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.

Figure 1 shows a simple example. Two isolates, *Ant* and *Tomcat*, are bound to the same policy for CPU time (CPU time domains are denoted by solid black lines) and heap memory (thick lines). Another isolate, *CloudDB*, is bound to separate domains for these resources. Thus *Ant* and *Tomcat* share allocations and constraints of these resources but *CloudDB*'s actions are confined to its own domains. Moreover, the three isolates are all bound to the same domain that limits the number of isolates they can collectively create (dotted line). For example, the policy may state that there cannot be more than four isolates within this domain, leaving room for one additional isolate.

The RM API does not itself 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 defined to execute prior to the consume event act as programmable *constraints* and can influence whether or not the request is granted. Consume actions defined to execute after the consume event can be thought of as *notifications*.

The following slightly simplified code shows how to create a domain for heap memory, with 32MB reserved, with a constraint that limits the use to no more than 32MB, and with a notification that gets triggered when the usage exceeds 30MB.

```
heapDomain1 = ResourceDomain.newDomain(HEAP_MEM);
heapDomain1.setReservation(32 * MEGABYTE);
heapDomain1.setConsumeAction(new Constraint() {
    public boolean consume(long previous, long proposed) {
        return (proposed <= 32 * MEGABYTE);
    }
});
heapDomain1.setConsumeAction(new Notification() {
    public void consumed(long previous, long current) {
        if (current > 30 * MEGABYTE)
            warn("Close to heap mem limit! Now using " + current);
    }
});
```

```
});
```

The RM API directly addresses the consumption of a given *quantity* of a resource but does not treat the *rate* of consumption as a resource in its own right. Instead, to impose a desired consumption rate for a given resource it suffices to throttle consumption requests until they match that rate.

The implementation of resources used by programs strictly encapsulates all interactions with the RM API. Thus when requesting a resource (e.g., opening a socket, etc.), clients are oblivious to the existence of the RM API and can be run under a particular resource management policy without any change to their code. Failures related to the enforcement of a particular policy are reported to the application as exceptions described in the resource's documentation.

3. Cluster Extensions

Load-balancers and application managers may need to directly control the distribution of isolates and to globally coordinate their node-local resource policies. Cluster-specific extensions, described below, are primarily intended for building such infrastructural components, but sophisticated applications can also take advantage of them. Most applications do not need to use any cluster-specific extensions (Sec. 5).

To support cluster computing, the Isolate API has been extended with the notion of an *aggregate*. An aggregate is a convenient way of naming a cluster node and abstracts away the details of spawning an instance of an isolate-enabled JVM on a specific node. For simplicity of exposition we assume that there is only one aggregate per node.

Aggregates are exposed to programmers as objects whose methods control the life-cycle of an aggregate (creation, activation, and termination), obtaining the current aggregate, and obtaining all known aggregates. The Isolate class has been extended with a method for creating an isolate in a particular aggregate. To give a flavor of programming with aggregates this code fragment starts a new isolate on each aggregate

```
Aggregate[] all = Aggregate.currentAggregates();
for (int i = 0; i < all.length; i++)
    new Isolate("tests.Hello", new String[] {}, all[i]).start();
```

Note that after creation isolates are started in the same way as in the API without the cluster extensions.

3.1 The RM API

Cluster extensions of the RM API fall into several categories. The first one allows for the creation of resource domains in remote aggregates:

```
domain = ResourceDomain.newDomain(resName, aggregate);
```

The result is the ability to programmatically control isolate creation and resource policy definition from any place in the cluster, without resorting to writing local proxies that would handle such operations.

The second group of extensions refines the notion of the scope of a given resource implementation. The key insight is that some resources have a source of manufacturing (i.e., implementation) specific to a given node; we call such resources *node-local*. Examples include CPU time, “manufactured” by processors specifically owned by a given computer, and heap memory, implemented on top of a virtual memory subsystem on a given machine. For node-local resources each node has its own bookkeeping RM API module, called from now on a *dispenser*¹ [8]. A dispenser for a node-local resource maintains information about consumption, reservations, constraints, and notifications pertaining to its node only. All domains for a given resource transparently consult the same dispenser before granting the resource. Upon getting a *consume request* a dispenser invokes the appropriate set of consume actions and reports their collective decision back to the requester.

Cluster-global resources generalize the notion of a single source of manufacture (i.e., the resource's implementation) to the whole cluster. Only one dispenser exists in the whole cluster for such resources. It maintains accounting information about a resource that is managed on a cluster-wide scale. Examples of cluster-global resources include traffic sent on inter-aggregate links and the total number of isolates. They facilitate explicit programmatic control over the combined resource consumption of distributed isolates.

Figure 2 shows the isolates from the previous figure executing on two cluster nodes (marked by thick lines). Domains for CPU time and network traffic are node-local, while the domain for the number of isolates allowed is cluster-global.

Each resource has to be declared as either cluster-global or node-local. For node-local resources, a binding of an isolate to a domain will succeed only if the isolate is created (locally or remotely) on node where the resource's dispenser executes. Domains for cluster-global resources have cluster-global scope, and any isolate can be bound to such a domain, regardless of where the isolate executes. The notions of node-local and cluster-global resources enable creation of application-specific resource management policies that span node boundaries, as explained in the next section.

¹Most applications do not see dispensers. Typically, only middleware, the JRE, or applications defining their own resources would explicitly create dispensers.

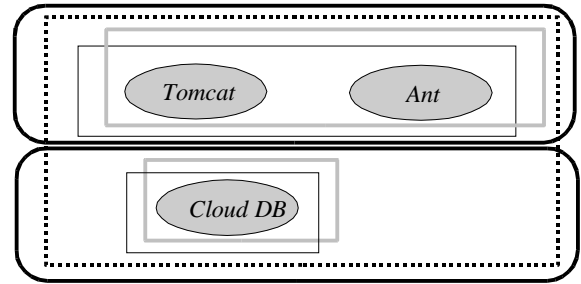


Figure 2. Isolates executing on two nodes and bound to resource domains.

After creation, cluster-global domains are used in the same way as the node-local ones.

The final addition to the RM API is an ability to query a given aggregate for its resource consumption and availability. This functionality parallels queries that can be issued against a specific resource domain. For example, a load balancer may need to verify that an aggregate has at least 1GB of heap memory available for a memory-intensive program:

```
long inUse = aggregate.getUsage(HEAP_MEM);  
long total = aggregate.getTotalQuantity(HEAP_MEM);  
return 1 * GB <= total - inUse;
```

4. Cluster-Global Policies

Cluster-global resources are useful in controlling the number of nodes a given application executes on or the number of isolates it has spawned, as the scope of these resources is inherently cluster-wide. To see the more general utility of cluster-global policies, consider a two-component application, consisting of isolates A and B. Further, consider network traffic, deployed as a node-local resource.

On a single node it is easy to control the combined usage of the network by both isolates – they simply need to be bound to the same resource domain for the resource. When the application is deployed on a cluster, with each component residing on a different computer, the following issue arises. The resource is node-local and it is not possible to create a domain that would allow for binding of mutually remote isolates to it. Thus A is subject to one policy and B to another and there is no correlation between the two policies unless explicitly programmed. Maintaining the accumulated count of usage of both components would require a custom application-level protocol.

Turning the outgoing network traffic resource from node-local to cluster global remedies the situation, in that it allows for global control of the resource. Technically this is straightforward: the proper deployment attribute has to be set for this resource before an aggregate is started. Since it is now a

cluster-global resource, the total usage by a given group of isolates can be controlled, regardless of how many nodes the groups spans, even though there may be multiple sources of manufacture of the resource (e.g., each node has its network card). Replacement of a collection of node-local domains with a single cluster-global one allows for expressing policies such as "the components of the application should never collectively exceed 1MB/s of out-bandwidth".

4.1 Dual view of a resource

When a node-local resource is turned into a cluster-global one, expressing global policies becomes relatively easy, but individual nodes lose the ability to control the resource locally. For example it is difficult to utter the following statement: "No component executing on this node can get more than .5MB/s of out-bandwidth". A related issue is that a policy that manages a resource on a global basis cannot detect shortages of resources locally; this can lead to a severely unbalanced load.

To address these issues the RM API allows each resource to be viewed in both ways: as node-local and as cluster-global. Thus each resource can be exposed through the RM API as two resource types, for example *ClusterGlobalNetworkTraffic* and *Node-LocalNetworkTraffic*. Physically, there is only one source of manufacture for the resource on any given node, but from the application perspective there are two distinct resources, each controlled by its own consumption policy, possibly independent from the policy that controls the other.

Figure 3 illustrates the dual view of a resource. The implementation of the resource (e.g., classes in java.net) on each aggregate transparently consults both dispensers upon request for the resource. The local dispenser has information about consumption on its aggregate, while the global one has a combined view of resource consumption on all the aggregates but does not distinguish between individual aggregates.

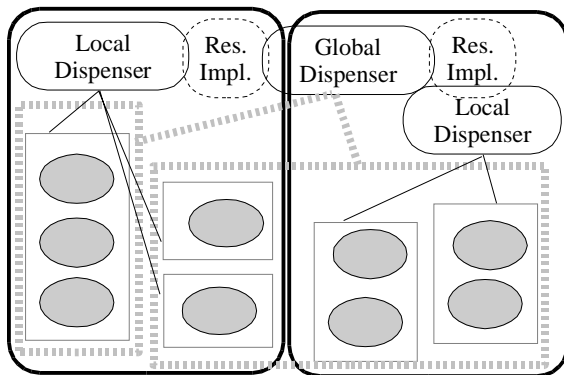


Figure 3. Scope of dispensers.

This mechanism enables both local and global control over a given resource. The policies may, but do not have to, coordinate. For example, the local policy may say "no more than 64MB for any isolate" while the global one may control memory allotments to groups of isolates stating that "no application, regardless of on how many nodes its components execute on, should get more than 1GB of heap memory". Each isolate would be thus controlled by both policies, and any request for the resource would be granted only if both policies agree. Other examples of policies that utilize both views of a resource are analyzed in Sec. 6.

It is important to note that explicit local and global policies are constrained by implicit physical local limits. For example, typically there is a finite number L of file descriptors available on any given node. A global policy applied to two mutually remote isolates that reserves 2L will guarantee that they *together* can use 2L, but will not guarantee that either of the isolates can individually obtain more than L descriptors.

5. Usage Notes

We have implemented the Isolate and RM APIs on top of a multi-tasking virtual machine [7]. MVM executes as a re-entrant virtual machine, and makes the runtime representation of loaded classes re-entrant as well. All immutable components of a class's runtime representation are shared across isolates. Because MVM aggressively shares meta-data its start-up time and memory footprint are significantly reduced. MVM can also execute isolate-unaware code.

The current implementation is based on the Java HotSpot™ Virtual Machine v. 1.3.1, client compiler. Our experimental platform (Sec. 6) consists of Sun Enterprise™ 420R servers with four UltraSPARC™ II processors and 4GB of main memory, with the Solaris™ 9 Operating Environment and connected by GigabitEthernet.

When executed on a cluster, each aggregate is implemented as an instance of MVM. Code written against the APIs presented in Section 2 will not be aware of the cluster. However, it will enjoy the benefits of clustering, as MVM provides several load-balancing strategies, selected by a start-up time option. Applications that need more control over isolate placement can utilize the APIs described in Section 3. Examples include custom load-balancers or code that exploits knowledge of the locality that may exist among the isolates that comprise an application (e.g., frequency of inter-isolate communication or accessing resources available on some aggregates only).

Aggregates are created in two ways: (i) indirectly, through a specified load-balancing strategy that creates fresh aggregates, and (ii) directly, either by using the Isolate API or by manually starting a new instance of MVM, which will find the rest of the cluster through a group membership protocol. Our current implementation uses JGroups [6]. Upon start-up MVM reads a configuration file that contains the names and attributes of resources to be managed.

MVM can tolerate aggregate failures (i.e., the remaining aggregates still function), and delivers lifecycle events so that applications can get immediate feedback on failures and take appropriate actions (e.g., restart the failed isolates in the remaining aggregates, re-adjust resource usage controls to shed load, or gracefully terminate the remaining aggregates). Due to the group membership protocol, the set of participating aggregates is dynamic.

Cluster extensions do not impact other aspects of MVM's functionality. In particular, we could not detect any performance loss for any of the SpecJVM98 benchmarks executed as remote isolates when compared to their local execution. The RM API provides programmable *granularity* of resource management [8] that controls the accounting cost vs precision trade-off. Selecting appropriate granularities leads to negligible (1% or less) overhead of management for all the resources that we have experimented with.

6. Experiments

The goal of the experiments is to show the types of cluster-wide resource policies expressible with the RM API. In the first experiment two isolates, V and W, execute on separate nodes. The resource controlled is outgoing network traffic. V is allowed to consume up to 4MB/s of the resource, and its usage varies over time. W continually tries to send out as much data as possible. W's consumption is subject to two policies: (i) local, which states that W cannot send more than

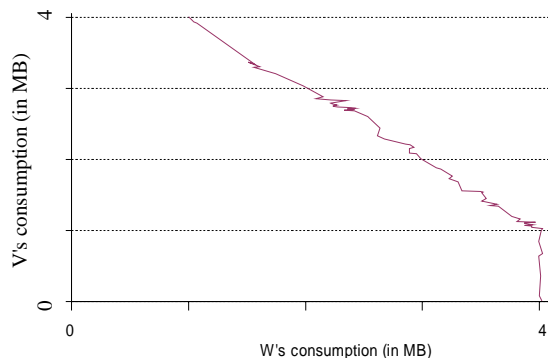


Figure 4. Constraining the network usage.

4MB/s of network traffic, and (ii) global, which ties W's allowed rate to V's consumption: the sum of V's and W's consumption rates cannot exceed 5MB/s. For example, when V's consumption rate is 0.5MB the global policy will cap W's usage at 4.5MB/s, which is more than the limit enforced by the local policy. Thus W will be allowed 4MB/s. When V's usage of the network rises to 3MB/s the global policy will determine a new limit value of 2MB/s, lower than the one dictated by the local policy. Hence, W will be able to use only 2MB/s. Finally, when V reaches 4MB W should be obtaining 1MB/s. Figure 4 shows the rates obtained by V (the Y axis) and W (the X axis). The effective rates obtained by both computations match closely the values specified by the policy.

The following code sketch shows how constraining W's usage of network bandwidth is orchestrated:

```

iV = new Isolate("tests.V", new String[] {}, agg1);
iW = new Isolate("tests.W", new String[] {}, agg2);
dL = ResourceDomain.newDomain(LOC_NET_OUT, agg2);
dG = ResourceDomain.newDomain(GLOB_NET_OUT);
dL.setConsumeAction(... constrain to no more than 4MB/s ...);
dG.setConsumeAction(... constrain to no more than 5MB/s ...);
dL.bind(iW);
dG.bind(iV);
dG.bind(iW);

```

The method arguments described in English are coded using pre-defined constraints that maintain rolling usage rates.

The second experiment, in which CPU time is controlled, is more complex. There are two dynamic, multi-isolate, synthetic, CPU-intensive applications, A and B, and two nodes (i.e., two aggregates). The policy consists of four clauses: (i) if both applications have isolates on both nodes, then each should get 50% of each node, (ii) if one application is present on one node only while the other has isolates on both nodes then the first one should get 80% of its only node, (iii) if an application is the sole one on a node it should get 100% of the node's CPU time, and finally (iv) each isolate should obtain an equal share of the CPU time within that application locally.

Figure 5 shows the progression of the experiment. Every 100s (the passage of time is plotted on the X axis) an isolate is started or terminated. The resulting CPU time allocations to particular isolates are plotted on the Y axis, separately for each aggregate's population of isolates. The plot is stacked: each isolate's CPU time usage rate is obtained by the value of its line minus the value of the line below it. For example, for aggregate 1 at $t=250$ there are two isolates from A: A:1 and A:3, and each has 50% of the CPU time on that machine.

Usage is reported as the amount obtained by an isolate in the most recent 20 seconds. This reporting

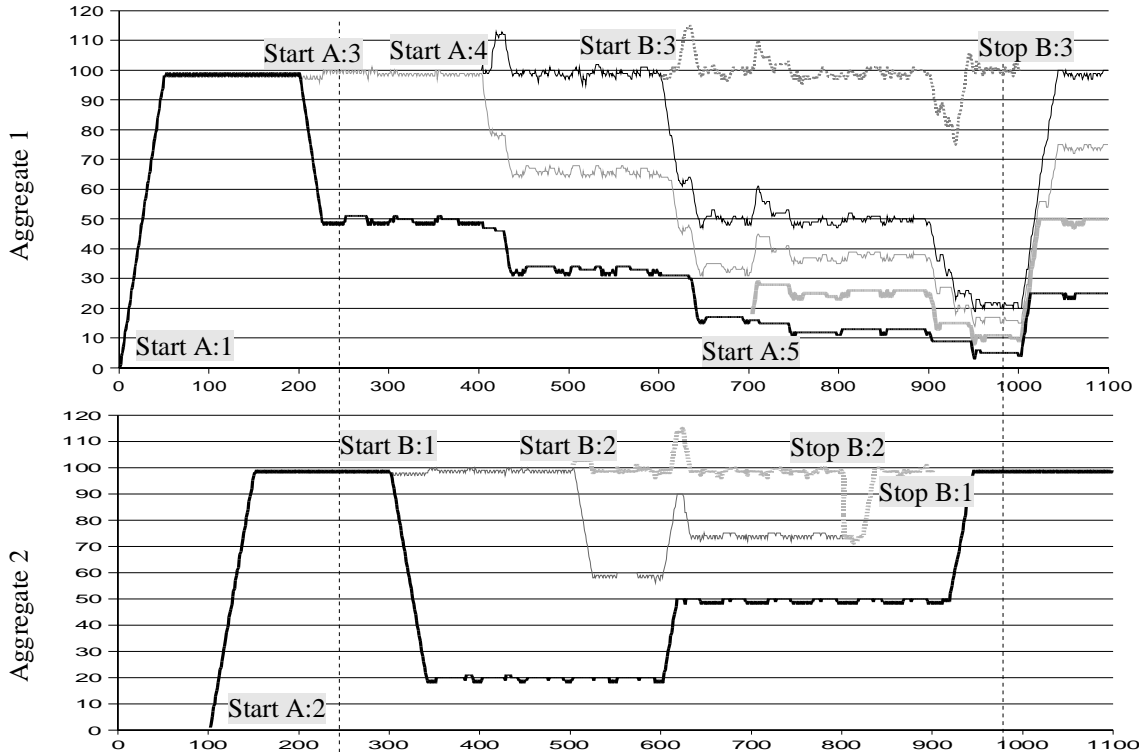


Figure 5. Consumption of CPU time by a dynamic set of isolates on two aggregates. $t=250$ and $t=975$ are marked by to help associate descriptions in the text with the actual consumption by individual isolates.

window manifests itself, for example, as a delay between an isolate's start and reaching its allowed rate.

Rates at which the isolates obtain CPU time match the policy limits well. Consider $t=975$. The one isolate left on aggregate 2, from A, has all of the aggregate's CPU time. On aggregate 1 there are five isolates. The only one from B is getting 80% of the resource while all the others are equally sharing the remaining 20%.

The experiments show that non-trivial fine-grained policies can be expressed with the RM API. The required global and local behavior programmed in resource consumption policies is accurately enforced. These and other scenarios can apply to other resources as well, either separately, when only a single resource needs to be controlled, or in multi-resource policies.

7. Related Work

Cluster reserves [3] are the closest in spirit and capabilities to the goal of our system. This abstraction aims at flexible and fine-grained allocation of CPU time to computations whose components execute on a cluster. Their policy engine uses a constraint solver to allocate tasks to nodes so that all the requirements are satisfied. A modified thread scheduler cooperates with cluster reserves to realize the imposed policy. The end result is that cluster reserves can obtain behavior

similar to our final experiment (Sec. 6). We believe that cluster reserves could be extended to operate on resources other than the CPU time and could use our techniques of bandwidth control through client suspension, without modifying schedulers. The deficiency of cluster reserves is the lack of programmability: CPU allocations are expressed at job submission time as numbers and are fixed throughout the duration of the task.

[8] contains a review of resource management systems for safe language platforms and their contrast with the RM API. Only one of them is based on a foundation of an isolated application [5], none possesses extensibility or programmability, and most are plagued by performance problems due to basing their accounting on bytecode rewriting. MVM's resource management capabilities are much more sophisticated than those in any of these systems.

Cluster-enabled JVM designs can be classified on several dimensions, and [9] is an excellent overview. From the programming perspective transparency is a vital characteristic. MVM places somewhere in the middle of the transparency axis. At one end there are single-image systems (e.g., [4, 2, 18, FFS04]), that provide a transparent and appealing programming model. However, obtaining scalability in these systems is hard, because of automatic conversion of

accesses from local to global, costs and performance may be severely degraded. Moreover, transparency interferes with failure handling, because it is hard to pinpoint and deal with failures when locations are hidden from the programmer. At the other end of the spectrum there are communication substrates [14, 11], sometimes equipped process life cycle management facilities. These are often harder to program because of special constructs and data types, and require manual management of resources. Our design takes a middle of the road approach to transparency: applications do not have to be aware of cluster topology, dynamics, or assignment of components to nodes, but distribution can occur only if an application is divided up into components or if multiple applications execute concurrently.

8. Summary

This paper describes key aspects of a design of a federation of JVMs forming a convenient infrastructure for executing multiple and/or multi-component Java applications on a cluster of interconnected computers. The design's focal point is comprehensive resource management applicable to cluster computing, compatible with the level of abstraction offered by modern object-oriented languages, and maintaining backwards-compatibility. Building on the foundation of a well-defined isolated component, the resulting resource management framework is capable of supporting a rich collection of resources and of defining policies that draw on all the features of the Java platform for their implementation.

The notion of a cluster-global resource naturally captures the aggregation of node-local resources, e.g., aggregated CPU time. This applies the familiar "single system image" attribute of clustered systems to the area of resource management. The ability to associate the multiple components (isolates) of a distributed application with a single resource domain (policy), dramatically simplifies resource accounting for cluster applications. The resulting platform makes managing resources for an application spanning multiple nodes no more difficult than controlling the resource consumption of programs executing on a single node. It forms a good basis for further exploration of the designs of cluster-operating environments based on safe languages.

Trademarks. Sun, Java, JVM, HotSpot, J2EE, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. UltraSPARC is registered trademark of SPARC International, Inc. in the United States.

9. References

- [1] Andersson, J., Weber, S., Cecchet, E., Jensen, C., Cahill, V. *Kaffemik: A Distributed JVM*. SCI-Europe, Dublin, Ireland, October 2001.
- [2] Antoniu, G., Bouge, L., Hatcher, P., MacBeth, M., McGuigan, K., and Namyst, R. *The Hyperion System: Compiling Java Bytecode for Distributed Execution*. *Parallel Computing*, 27(10), 2001.
- [3] Aron, M., Drushel, P., and Zwaenepoel, W., *Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers*. ACM Sigmetrics, Santa Clara, CA, June 2000.
- [4] Aridor, Y., Factor, M., and Teperman, A. *cJVM: A Single-System Image of a JVM on a Cluster*. International Conference on Parallel Processing, Fukushima, Japan, September 1999.
- [5] Back, G., Hsieh, W., and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th OSDI, San Diego, CA, 2000.
- [6] Ban, B. *JGroups – A Toolkit for Reliable Multicast Communication*. <http://www.jgroups.org>.
- [7] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution*. 17th ACM OOPSLA'01, Tampa, FL, October 2001.
- [8] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. *A Resource Management Interface for the Java Platform*. *Software – Practice and Experience*. 35 (2) 123-157 (2005).
- [9] Factor, M., Schuster, A., Shagin, K. *A Distributed Runtime for Java: Yesterday and Today*. 18th International Parallel and Distributed Processing Symposium, Santa Fe, NM, April 2004.
- [10] Foster, I., Kesselman, C. (editors) *The Grid 2: Blueprint for a New Computing Infrastructure*. 2nd Edition. Morgan Kaufmann, 2003.
- [11] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine*. MIT Press, November 1994.
- [12] Java Community Process. JSR 121: Application Isolation API. <http://jcp.org/jsr/detail/121.jsp>.
- [13] Jordan, M., Czajkowski, G., Kouklinski, K., and Skinner, G. *Extending a J2EE Server with Dynamic and Flexible Resource Management*. ACM/IFIP/USENIX Middleware Conf., Toronto, ON, Oct. 2004.
- [14] Message Passing Forum. *MPI: A Message-Passing Interface Standard*. Technical Report UT-CS-94-230, University of Tennessee, 1994.
- [15] Philippsen, M., and Zenger, M. *JavaParty – Transparent Remote Objects in Java*. *Concurrency: Practice and Experience*, 9(11): 1225-1242, 1997.
- [16] Tilevich, E., Smaragdakis, Y. *J-Orchestra: Automatic Java Application Partitioning*. ECOOP'02, Malaga, Spain, June 2002.
- [17] Yu, W., and Cox, A. *Java/DSM: A Platform for Heterogeneous Computing*. *Concurrency – Practice and Experience*, 9(11), 1997.
- [18] Zhu, W., Wang, C-L., and Lau, F. *Jessica2: A Distributed JVM with Transparent Thread Migration Support*. IEEE Cluster'02, Chicago, IL, Sept. 2002.