Resource and System Management for Server Farms

Ignacio Laguna, Gunjan Khanna
Dependable Computing Systems Lab
Purdue University

Information copied from the following papers:

- ***Extending a J2EE Server with Dynamic and Flexible Resource Management***. M. Jordan, G. Czajkowski, K. Kouklinski, G. Skinner, Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, 2004.
- ***Scaling J2EE Application Servers with the Multi-Tasking Virtual Machine***. Jordan, M., Daynes, L., Czajkowski, G., Jarzab, M., Bryce. C., ACM, Software—Practice & Experience, Volume 36 , Issue 6 (May 2006).
- ***Multitasking without comprimise: a virtual machine evolution.*** G. Czajkowski and L. Daynés. Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, 2001.

Outline

Introduction

- The Java™ 2 Platform, Enterprise Edition (J2EE) is an standard server-side environment for developing enterprise applications in Java.
- The combination of the J2EE and J2SE platforms act as the underlying operating system (OS) for e-commerce applications.

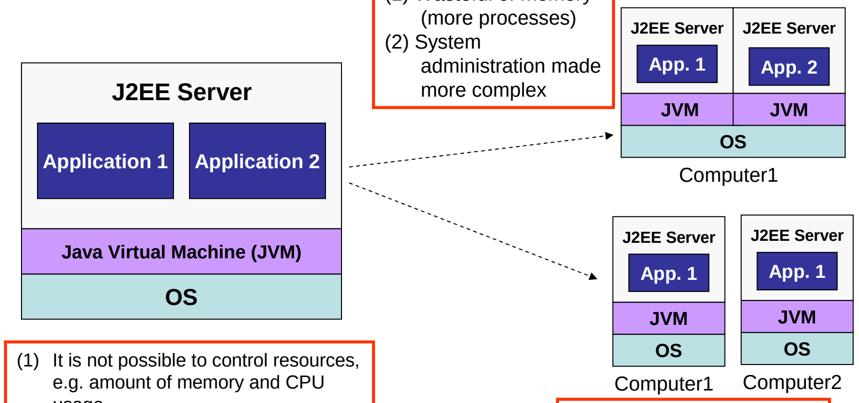
  For example:
  - A single J2EE server can host several applications, possibly from different organizations, that must compete for the server resources.
- In practice, J2SE/J2EE platforms omit some important features that are standard in OS → J2EE applications can not properly isolated.

Introduction (cont.)

- Capabilities of resource/system management is notably **absent** in current JVM implementations.
- Ad-hoc methods to exploit the OS capabilities limit the **portability** of the platform.
- J2EE platforms defines other kind of sources with no analog counterparts in an operating system environments.

  Example: # of JDBC connections available to an application.

Limitations in current server isolation methods



(1) Wasteful of memory (more processes)
(2) System administration made more complex

(1) It is not possible to control resources, e.g. amount of memory and CPU usage.
(2) Can not terminate threads safely

Expensive and inefficient in terms of hardware.

Isolates and the Multi-tasking VM

- In previous work, It has been described a extensible framework for resource management expressed entirely in the Java language.
- This framework (API) is able to handle efficiently traditional resources:
  - CPU time, Memory, Network, Programmed-defined resources (e.g., JDBC connections)
- This API introduces the concept of *isolates*: a fundamental unit of accounting.
- Resource management (RM) is based on isolates.

Isolates and the Multi-tasking VM

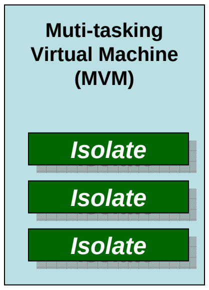
- An ***isolate*** is a Java application component that does not share any objects with other isolates.
- The Isolate API allows for the dynamic creation and destruction of isolates.
- Example of creation of an isolate, that will execute my Myclass with a single argument “abc”:

  new Isolate(“MyClass”, new String[ ] {“abc”}).start();

- Isolates does not share objects, but they can communicate using traditional inter-process communication: sockets and files.

January 17, 24 (2007) 7/25

Isolate characteristics



• Does not share objects with other isolates.

• It is possible to pass contextual information to an isolate, required when starting an application.

• Can communicate using inter-process mechanisms (e.g., sockets and files)

• Isolate API is fully compatible with existing applications. (There is no need to modify app. unaware of the API)

*• The current implementation allow to **suspend** and to **resume** isolates (useful for CPU consumption policies).*

• One implementation is for all isolates to reside in a single instance of the JRE (called MVM).

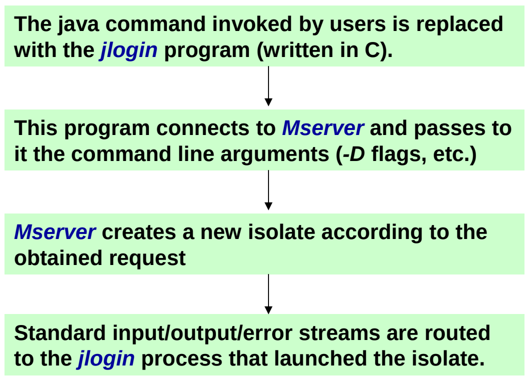
January 17, 24 (2007) 8/25

Multi-tasking Virtual Machine (MVM)

- MVM is a general-purpose virtual machine for executing multiple applications written in Java.
- MVM transparently shares significant portions of the virtual machine among isolates. For example:
  - Run-time representations of all loaded classes and compiled code are shared.
- In effect, each application “believes” it executes in its own private JVM.
- Certain JRE classes had to be modified (**System** and **Runtime**) to make some operations apply only on the calling isolate (example: **System.exit()**).

Operation of the MVM

The first isolate is a simple application called ***Mserver*** that listens on a socket for connections.



The java command invoked by users is replaced with the *jlogin* program (written in C).

This program connects to *Mserver* and passes to it the command line 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.

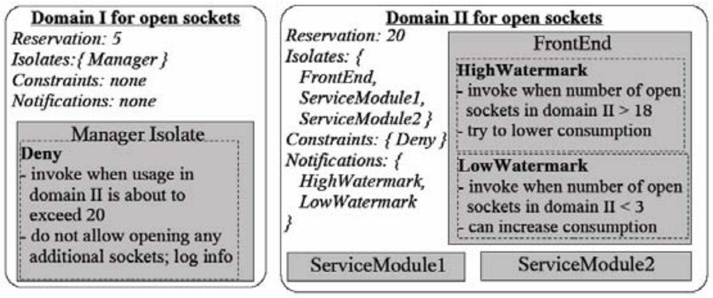
Resource Management API

- Resources can be exposed through the RM API in a uniform way.
- 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.
- 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.

January 17, 24 (2007) 11/25

Resource Management API example

- 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.

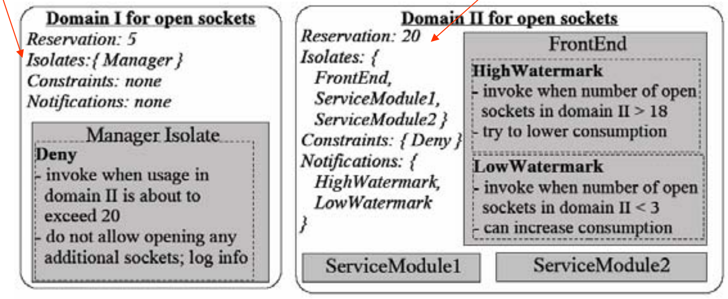

Domain I for open sockets
Reservation: 5
Isolates: { Manager }
Constraints: none
Notifications: none
Manager Isolate
Deny
- invoke when usage in domain II is about to exceed 20
- do not allow opening any additional sockets; log info
Domain II for open sockets
Reservation: 20
Isolates: { FrontEnd, ServiceModule1, 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

January 17, 24 (2007) 12/25

Resource Management API example

A *Manager* isolate executes in its own resource domain.

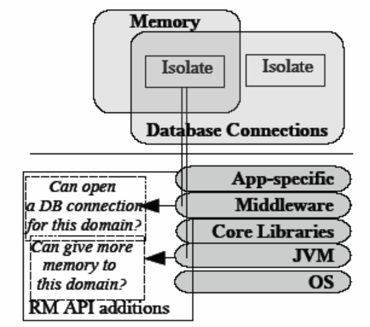
The second domain has a reservation of 20. This quantity is guaranteed (but only if there are no constraints limiting the resource's availability).


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

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.

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 o the resource available to resource domains and thus (indirectly) to isolates.
- Dispensers do not themselves implement resources.



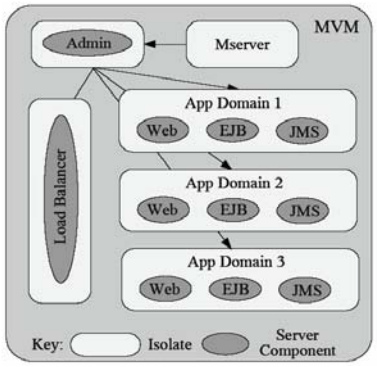
Resource implementations consult their dispensers when granting a request. The decisions depend on policies defined in resource domains.

J2EE on MVM

- Isolates in this research were implemented using the J2EE 1.3.1 Reference Implementation (J2EERI).
- J2EE specification defines four application types and an array of services (e.g., JMS and JDBC). 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.
- An ***application domain*** hosts one (or more) complete J2EE applications, and the associated containers and servers.

J2EE on MVM

- Application domain isolates are controlled by a separate administrative isolate.
- 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 can be performed with respect to arbitrary resources, but is typically limited to CPU load and network traffic.
- It uses the usage information provided in the notification to maintain a load average for each domain.



J2SE Resources

- In this research they explored the management of **CPU time** in the J2EERI server.
- 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:
  - but it does have the property that CPU is actually used before the consume action can impose control.
- A rate-adjusting policy is specified as a given **threshold** value in a given time period, e.g., 100 units in 1 second.
- The policy maintains a history of resource usage over the interval:

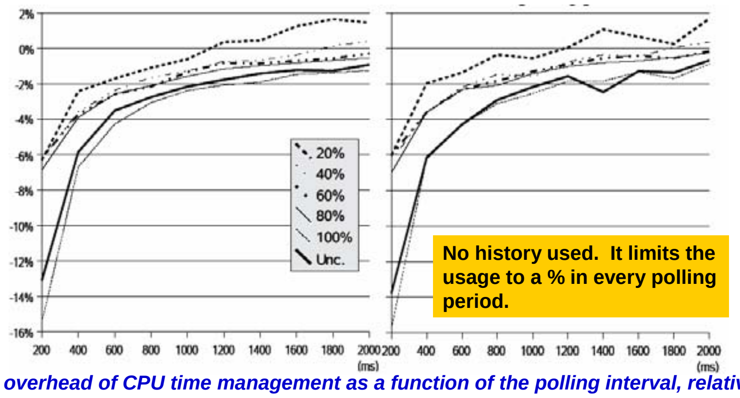
```
if consumption > value, suspends the isolate for ∆t
```

Throughput measurements in RM

- Formula used to compute the potential length of suspension:

  U * (I / T) – H

  An interval *I*, a threshold *T*, history interval *H*, and usage in history interval *U*



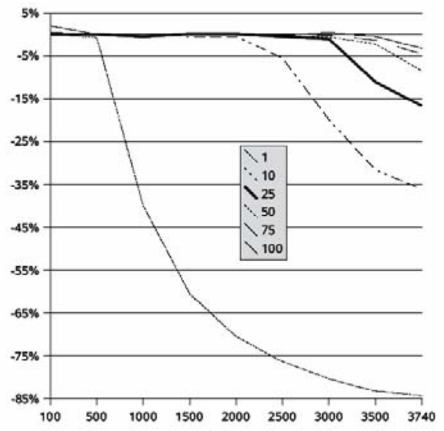
The overhead of CPU time management as a function of the polling interval, relative to a server with no RM enabled (lower is worse).

J2EE-specific implemented resources

- Resources chosen was related to database management, namely connections, statements, and transactions.

- **Connections:**
  - A fundamental JDBC resource is a *connection*, which represents an active session with the database server.
- **Transactions:**
  - Requires action on part of the database server to make relevant data durable, which typically requires disk activity.
- **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.

Experiments with J2EE-specific resources

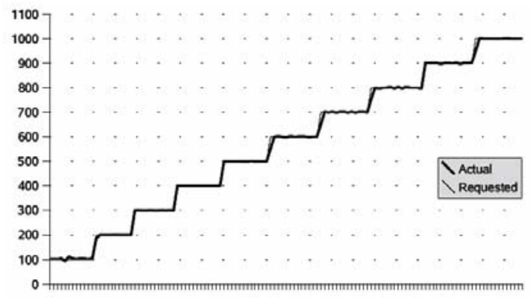


Percentage difference in actual transaction rate against expected transaction rate for different resource granularities.

- ***Granularity*****:** the minimal indivisible amount of the resource in a given implementation.
- When run with no constraints on transaction rate, the benchmark achieves 3740 transactions per second.
- A granularity of 1 adds considerable overhead
- At granularities of 50 and above, the maximum reduction is 8.5%

Experiments with J2EE-specific resources (cont.)

- This plots the measured and requested transaction rates against time for a subset of the transaction rate range and a granularity of 100.
- It shows that the measured rate closely tracks the requested rate.



Responsiveness of controlling the rate of transactions.

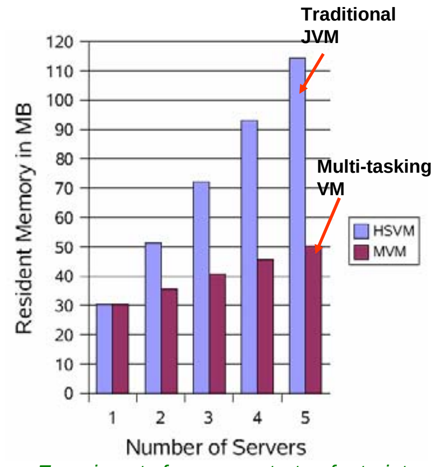
Summary

- 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.
- It significantly reduces the complexity of provisioning resources for code deployed in application servers. Allows for precise and inexpensive usage monitoring.
- Current prototype is already satisfactory for practical use, with very low – in some cases not measurable – overheads.

Appendix

Memory footprint experiments in the Multi-tasking Virtual Machine

- Compare **MVM** with **Java HotSpot Virtual Machine** (better performance for most applications).
- Only **resident (physical)** memory associated with process is counted.
- Additional **virtual memory** may have been reserved (but not counted).
- In the tests, **HSVM** used the standard heapsize default (i.e, max 64MB).
- For **MVM**, it had a max heap size of 128MB and 8 isolates.



Experiments for server startup footprint

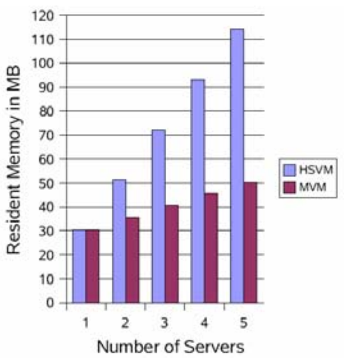
Appendix (cont.)

- No significant difference for one server

- A reduction of 31% for 2 servers, and 43% for 3 servers.

- Incremental overhead (per server):

HSVM: 20MB
MVM: 5MB



- MVM sharing class metadata mechanisms is proportionally more effective.

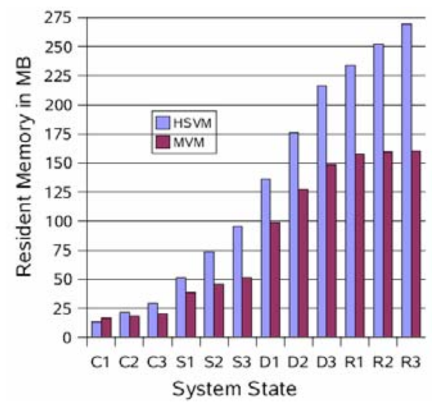
Data: allocated objects.

Metadata: bytecodes of methods, compiled bytecodes, other memory.

VM code: the only portion that can be shared by the OS (but is small).

Memory %	*S1*	*S2*	*S3*	*S4*	*S5*
Data	20.06	29.62	37.13	41.41	45.43
Metadata	57.38	51.1	46.1	43.58	40.84
VM code	22.56	19.28	16.76	15.01	13.63

Appendix (cont.)



Footprint with deployed Petstore application

- **An application (Petstore) was deployed and run in each server.**

- **It required a SQL database (Cloudscape).**

- **Key of x-axis:**

Cn: Cloudscape (DB) server n started

Sn: J2EE server n started

Dn: Petstore deployed on server n

Rn: Petstore run on server n (a single item is purchased using a web browser).