State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine[1]

Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth,
Gregory A. Hill, and Raul Saavedra
Institute for Human & Machine Cognition, University of West Florida
{nsuri,jbradshaw,mbreedy,pgroth,ghill,rsaavedra}@ai.uwf.edu

Extended Abstract

Although Java is currently riding a rising wave of popularity, current versions fail to address many of the unique challenges posed by the new generation of distributed applications. In particular the advent of peer-to-peer computing models and the proliferation of software agents motivates various requirements that go beyond the capabilities of current Java Virtual Machines:

- *Full state capture*. To support checkpointing and load balancing, the Virtual Machine (VM) must be able to capture its complete state including all threads, objects, and classes in the heap. To support requirements for strong "anytime" mobility and forced migration (such as when a host is about to go offline), the VM must be able to support asynchronous requests to capture execution state for a thread or thread group.

- *Dynamic access and resource control.* The security model in Java 2 [4] provides a fairly comprehensive model for access control but does not allow for dynamic permission revocation. Once permission is granted to a process, that permission is in effect for the lifetime of that process. Furthermore, there is no way of specifying the amount of a resource that is granted, assuring a specific quality of service for each process. For example, one would like to be able to limit the quantity of hard disk, network, or CPU usage that is available to a given process or to determine the rate at which the resource may be used. Denial-of-service conditions on a host or network resulting from code that is poorly programmed, malicious, or has been tampered with are impossible to detect and interrupt without dynamic monitoring and control mechanisms for individual processes.

- *Resource accounting.* Tracking of resource use, based on resource control mechanisms, enables accounting and billing mechanisms that hosts can use to calculate charges for resident programs. The same mechanisms can be used to detect patterns of resource abuse.

The Aroma VM is a Java-compatible VM that provides unique capabilities such as thread and VM state capture and dynamic, fine-grained resource control and accounting. Aroma was developed as part of a research project on mobile agent systems and distributed systems. The overall goal was to develop a VM for research use that would be simple, flexible, and portable. Therefore, Aroma tries to minimize dependence on operating systems features and avoids platform-specific assembly code. Aroma is currently being used as part of the NOMADS mobile agent system and in the WYA (While You're Away) distributed system for load balancing and utilizing idle workstations.

Aroma currently provides mechanisms to capture the state of individual threads or the complete VM. Individual thread state capture examines each of the stack frames in the method stack of the thread and saves all relevant values (such as the program counter, the local variables, and the operand stack) as well as all reachable objects. Full VM state capture saves the state of all threads in the VM as well as all loaded classes. State capture may be initiated synchronously by a thread or asynchronously by an external request.

Java threads in Aroma have a one-to-one mapping to native operating system threads. The primary reason for mapping directly to native threads was to not have a platform-specific user-level threads package. However, mapping to native threads complicates the task of capturing a Java thread's state. In particular, asynchronous

[1] This research is supported in part by DARPA's Control of Agent-Based Systems (CoABS) program and the National Technology Alliance (NTA) / National Imagery and Mapping Agency (NIMA)

requests to capture thread state are harder to support because threads may be in one of many states (such as running, blocked, waiting, sleeping, or suspended) when the state capture is requested.

Several components of Aroma were carefully designed to support state capture of asynchronous Java threads. The Java threads and their corresponding native threads are decoupled so that the native thread's stack stays constant while allowing the Java thread's stack to change with method invocations and returns. Aroma also uses special stack frames on the Java thread stack to handle situations where Java code and C code might be interleaved (such as a Java method instantiating an object which requires native code to load the class which might again require a Java method to be invoked to initialize the class). Finally, the monitors in Aroma were carefully designed to be abortable so that the state of Java threads can be captured even if they are blocked trying to enter a monitor or waiting on a condition variable.

Aroma currently provides a comprehensive set of resource controls for CPU, disk, and network. The resource control mechanisms allow limits to be placed on both the rate and quantity of resources used by Java threads. Rate limits include CPU usage, disk read rate, disk write rate, network read rate, and network write rate. Rate limits for I/O are specified in bytes/millisecond. Quantity limits include disk space, total bytes written to disk, total bytes read from the disk, total bytes written to the network, and total bytes read from the network. Quantity limits are specified in bytes.

CPU resource control was designed to support two alternative means of expressing the resource limits. The first alternative is to express the limit in terms of bytecodes executed per millisecond. The advantage of expressing a limit in terms of bytecodes per unit time is that given the processing requirements of a thread, the thread's execution time (or time to complete a task) may be predicted. Another advantage of expressing limits in terms of bytecodes per unit time is that the limit is system and architecture independent. The second alternative is to express the limit in terms of some percentage of CPU time, expressed as a number between 0 and 100. Expressing limits as a percentage of overall CPU time on a host provides better control over resource consumption on that particular host.

Rate limits for disk and network are expressed in terms of bytes read or written per millisecond. If a rate limit is in effect, then I/O operations are transparently delayed if necessary until such time that allowing the operation would not exceed the limit. Threads performing I/O operations will not be aware of any resource limits in place unless they choose to query the VM.

Quantity limits for disk and network are expressed in terms of bytes. If a quantity limit is in effect, then the VM throws an exception when a thread requests an I/O operation that would result in the limit being exceeded.

The Aroma VM implementation is based on the Java Virtual Machine specification and does not use any source code from other licensed VM implementations. Therefore, Aroma may be distributed without any licensing constraints. Currently, Aroma is distributed in binary form as bundled with the NOMADS mobile agent system. NOMADS may be downloaded and used free of charge for non-commercial purposes from http://www.coginst.uwf.edu/nomads. We also plan to release the Aroma VM in the form of an object library that may be embedded inside other applications.

Aroma is currently JDK 1.2.2 "compatible" with some missing features such as support for AWT and Swing. Also, Aroma currently works with the Sun implementation of the Java Platform API (as distributed in the Java Runtime Environment). In the future, we also plan to support other API implementations such as the GNU Classpath project. Aroma has been ported to Win 32 on x86, Solaris on SPARC, and Linux on x86.

Currently, Aroma does not provide a Just-in-Time compiler, which significantly affects the performance of Aroma when compared with other VM implementations. In the future, we will work on integrating freely available JIT compilers (such as OpenJIT) while still retaining the unique features of Aroma. The VM does offer good state capture performance and has minimal overhead for disk and network resource controls. CPU resource control introduces an overhead of 6.6% to 6.9%. More information on the Aroma VM, the NOMADS mobile agent system, and the WYA (While You're Away) distributed system is available on the NOMADS web site at http://www.coginst.uwf.edu/nomads.