JSR-174 - Requirements for the Monitoring and Management Specification for the Java^{TM} Virtual Machine.

PUBLIC DRAFT 0.99.7 (25-Sep-03)

Contents

1 Abstract 5
2 Motivation & Goals 5
3 Target Audiences 6
3.1 Internal Monitoring and Management 6
3.2 External Monitoring and Management 6
4 Relationship of JSR174 with JSR163 7
5 Convention 7
Part 1. General Requirements 8
6 Main Characteristics 8
6.1 Monitoring Model 8
6.2 Low Overhead 8
6.3 On-demand Monitoring 8
6.4 24x7 Operations, No Restart 8
6.5 Multiple Management Clients Support 8
6.6 Low Memory Detection Support 8
6.7 Deadlock Detection Support 9
6.8 Remote JVM Monitoring and Management Support 9
6.9 SNMP Support 9
6.10 Minimise "Heisenberg Principle" Effects for Application Self-monitoring 9
6.11 Extensions Mechanism 9
7 JVM Data for Monitoring and Management 9
7.1 OS Resources 9
7.2 Operating System, Runtime and JIT Compiler Properties 10
7.3 Compilation, Execution and Synchronisation Subsystems 10
7.4 Class Loading Subsystem 12
7.5 Memory Subsystem 12
Part 2. Java Language Application Programming Interface 17
8 Requirements for the Java API 17
8.1 Production Environment 17
8.2 Access to JVM Monitoring Data 17
8.3 On-Demand Monitoring 17
8.4 24x7 Operations, No Restart 17
8.5 Multi-Thread Safe 17
8.6 Optional Data/Operation 17
8.7 Extensible Mechanism 17
8.8 JMX MBeans 17
Part 3. Native Interface 18
9 General requirements for the Java native interface 18
9.1 Dynamic Load of Native Agents and Late Binding 18
9.2 Support for Multiple Concurrent Agents 18
9.3 24x7 Operation without Restarting the JVM 18
9.4 Extensions capabilities 18
9.5 Rules for writing agent code 19
10 Additional requirements 19
10.1 Operating System, Runtime and JIT Compiler Properties 19
10.2 Execution and Synchronization Subsystem 19
10.3 Class Loading Subsystem 19
10.4 Memory Subsystem 20

Part 4. Appendixes 21
Appendix A. Expert Group Collaboration Agreement 21
Appendix B. Trademarks 22

Tables

Table 7.2-1 OS and Runtime Properties 10
Table 7.3-1 Compilation, execution and Synchronization Subsystem 11
Table 7.4-1 Class Loading Subsystem 12
Table 7.5-1 Memory Subsystem 16
Table 10.1-1 Native Interface - OS, Runtime and JIT properties 19

1 Abstract

This document describes the requirements for the Monitoring and Management Specification for the Java™ Virtual Machine (JVM™)[1]. JSR174 requires that both the native interface as well as the Java programming language interface be defined for JVM monitoring and management. This document is structured as three technical parts plus two administrative appendixes. The technical parts are:

- Part 1 of this document describes, in a generic and platform neutral style, all the requirements applicable to both the Java programming language interface and the native interface.
- Part 2 describes the requirements specific to the Java programming language interface.
- Part 3 describes the requirements specific to the native interface.

This specification is intended to be used as guidelines for the development requirements of the monitoring and management portion of the JVMTI specification and related API being proposed by the JSR-163 "Java™ Platform Profiling Architecture".

2 Motivation & Goals

Modern enterprise solutions are complex and often comprise a stack of different distributed products, many of which are Java platform applications. In such an environment, the task of monitoring and managing the solution is very complex and requires a collection of probes and adapters in order to collect the necessary data.

Monitoring and understanding the overall health of the JVM is an important contributor to the management of the solution, and the monitoring should be as non-intrusive as possible in order not to alter the original characteristics of the solution.

JVM monitoring and management will be most useful in customer production environments where it will be used to gather information about the health of the JVM and the application and to dynamically select/configure data collection.

The current specification for the JVM does not include any lightweight standard interface for the monitoring of its health indicators nor for the management of some of its run-time characteristics.

The main goal of this specification is to provide the necessary guidelines and recommendations for the definition of a set of APIs to enable the monitoring and management of the JVM. This specification will _not_ propose any

[1] The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java™ platform.

interfaces to monitor or manage applications, nor to supply a higher level abstraction to monitoring and management. Monitoring and management intelligence shall be provided by another layer yet to be defined. The expectation is that monitoring and management intelligence shall be provided by 3rd party software vendors.

In essence, JSR174 is about the definition of the data required for the lightweight monitoring and the management of the Java virtual machine.

3 Target Audiences

The target audiences of the applications using JVM data for monitoring and management fall into the following groups:

- Internal monitoring and management
- External monitoring and management

3.1 Internal Monitoring and Management

Java platform applications require JVM data for self monitoring and management of the running JVM.

3.2 External Monitoring and Management

The target audiences of this group are as follows:

3.2.1 Management systems, end users and administrators

Management systems require the JVM data to help end users and administrators to perform continuous production monitoring of an application in the deployed environment.

3.2.2 IT organization and Support Engineers

IT organization and support engineers will use the JVM data to help identify and troubleshoot any problems occurring in the production environment. These could be configuration issues, performance bottlenecks or bugs in the applications. Support engineers can also use JVM implementation specific data to better understand the behaviour of a specific JVM implementation.

3.2.3 Development organization

When a problem occurs in the production environment, it may be necessary to involve the development organization to help with diagnosis. The developers often request more detailed JVM data from the production environment. Such data may be very specific to the JVM implementation.

4 Relationship of JSR174 with JSR163

The Expert Groups for JSR 174 and JSR 163 have agreed that due to the functional overlap and target audience of the two JSRs, they should share the same infrastructure for surfacing as well as producing the monitoring data and managing the JVM.

- JSR-174 will define the requirements for JVM monitoring and management. These requirements will be incorporated into JSR-163.
- JSR-174 will pre-requisite JSR-163 JVMTI (with requirements).
- JSR-163 will provide the interface, Reference Implementation and Test Compatibility Kit to meet the needs of monitoring, management and profiling.

5 Convention

The monitoring data and metrics are classified as follows:

- Mandatory

  Data and metrics are fully specified and supported by all JVM implementations that conform to the specification.

- Optional

  Data and metrics are fully specified but may only be supported by some JVM implementations. A capabilities query mechanism shall be used to determine which data and metrics are supported. Optional data allows users to easily build a modular system to support multiple JVM's.

- Platform Specific

  Data and metrics are not specified in this document. An extensions mechanism shall be used to query a given JVM for its platform specific support.

Part 1. General Requirements

The two most important characteristics for monitoring of production environments (24x7 operations) are:

- Non-intrusiveness (or as lightweight as possible)
- Low data rate

Monitoring should be used to obtain some indication of how well the JVM is functioning and should provide the necessary data to be used to trigger further investigation. Anything beyond that should be covered by other activities such as profiling and/or debugging and/or problem determination.

The specification shall define both mandatory and optional requirements for monitoring and management of a JVM implementation.

6 Main Characteristics

6.1 Monitoring Model

The monitoring of low frequency events is usually associated with synchronous notifications while high frequency events are usually summarized in counters and obtainable through polling.

6.2 Low Overhead

There should be no more than 1% performance overhead as measured by standard benchmarks (specJBB, specJVM, ECPerf, etc).

6.3 On-demand Monitoring

Event generation and counters can be enabled/disabled/reset on demand.

6.4 24x7 Operations, No Restart

No restart shall be required for attaching/enabling/disabling/resetting the monitoring.

6.5 Multiple Management Clients Support

Concurrent monitoring and management by more than one client shall be supported.

6.6 Low Memory Detection Support

A mechanism to allow an application to detect a JVM in a low memory situation shall be provided.

The mechanism is not intended to allow an application to manage itself and recover from a low memory condition, for example to free up memory used by the application. Instead, the kind of action an application or its external agent may take would be to stop distributing any work to a JVM when it detects that JVM is in a low memory situation. Furthermore, there is no intent to define timely notification, due to the dynamic nature of a Java platform application.

6.7 Deadlock Detection Support

A mechanism and necessary monitoring data to allow an application or management client to detect application deadlock shall be provided.

6.8 Remote JVM Monitoring and Management Support

Java™ Management Extension (JMX) MBean for the monitoring and management of the JVM shall be provided.

6.9 SNMP Support

An SNMP MIB definition to enable SNMP management applications to monitor and manage the JVM shall be provided.

6.10 Minimise "Heisenberg Principle" Effects for Application Self-monitoring

Special consideration shall be given to the implementation of the Monitoring and Management API in order to minimise “Heisenberg Principle” effects.

6.11 Extensions Mechanism

A mechanism to allow JVM vendors to provide extensions for platform specific monitoring data and management capabilities shall be provided.

The JVM Tools Interface currently being specified on JSR-163 has the necessary provision for extensions by vendors.

7 JVM Data for Monitoring and Management

This chapter describes and categorizes the requirement for data and metrics for the JVM health indicators that are to be monitored.

7.1 OS Resources

There has been a growing interest in surfacing some level of information about the OS resources a given JVM is consuming as well as changes in the available OS resources. It might become a little challenging to create the necessary common abstraction to map OS resources from the various OS in the market. Our recommendation is to use as much as possible the

infrastructure provided by the OS to monitor the required data but have the necessary mechanism to allow platform specific extensions.

7.2 Operating System, Runtime and JIT Compiler Properties

This category comprises the information needed to identify the Operating System platform, the JVM and JIT Compiler implementations, the command line options and arguments.

Mandatory	**Optional**
OS Properties • os.name • os.version • os.arch JVM system properties • java.vm.vendor • java.vm.version • java.vm.name • java.vm.specification.version • java.vm.specification.vendor • java.vm.specification.name • java.class.path • java.library.path Compilation/JIT properties • name of the compiler JVM Startup options	Boot class path

Table 7.2-1 OS and Runtime Properties

7.3 Compilation, Execution and Synchronisation Subsystems

Monitoring data from the execution and synchronisation subsystems provide valuable information about the state, properties, counting of threads, thread CPU utilization, JIT compilation time and also valuable information for the identification of deadlocks and hot-locks.

First of all, we should keep in mind that there are different implementations of the JVM *threading and locking infrastructure* and the cost for providing thread/locking enumeration/timing/counting may vary considerably. We would like the provision of some sort of object locking (a.k.a. monitor lock) statistics that would enable us to identify that contention is taking place.

A thread will be uniquely identified during its existence. A given thread's identifier might be reused after a given thread terminates. it will be the responsibility of the monitoring agents/code to differentiate a thread that has been destroyed from a newly created one.

Mandatory	**Optional**
Counters and accumulators • Total number of Java threads created and started since the JVM started • Current number/list of live Java threads • Peak number of live Java threads • Current number of Daemon threads	

Thread general info
 	
• Thread Identifier (may not be unique over the lifetime of a JVM run)
• Thread state (JVM thread state, not necessarily OS thread state)
    o Created but not-yet-started
    o Running
    o Blocked on entering a monitor
    o Blocked waiting on a monitor
    o Suspended
    o Terminated
    o Running on JNI code
    o Undefined
• The monitor a thread is blocked on, if any.
• The thread which owns the monitor lock for a given object
• Total number of contentions (number of times a given thread was blocked to enter or waiting on a monitor.)
• Thread stack trace
	Thread general info
	
	• Thread state - waiting on I/O
	• Total approximated accumulated elapsed time spent by given thread that blocked on monitors or waited on monitors (Disabled by default)
	• Total CPU time consumed by a given thread.

JVM Uptime • Approximate total time since the JVM started	JIT Compilation Time • Approximate total accumulated time spent in JIT compilation since the JVM started
	Manageable attribute • Enable/disable thread contention monitoring • Enable/disable thread CPU time measurement.

Table 7.3-1 Compilation, execution and Synchronization Subsystem

7.4 Class Loading Subsystem

Monitoring data from the class loading subsystem can be grouped on a *class loader basis* or on a *class basis*. Either way should provide the necessary data to allow a management application to infer the health of the class loading subsystem. This JSR will specify *class basis* grouping as the general requirement.

Mandatory	**Optional**
Counters and accumulators • Current number of non-array classes loaded in the JVM • Total number of non-array classes loaded since the JVM started. • Total number of non-array classes unloaded since the JVM started. Manageable attributes • Turn on/off verboseness	

Table 7.4-1 Class Loading Subsystem

7.5 Memory Subsystem

The memory subsystem is responsible for managing and accounting for all the memory in use by the JVM. As the implementation of the memory subsystem varies from vendor to vendor, we propose a common abstract model for describing allocated memory and the behaviour of the memory subsystem.

The JVM memory subsystem manages all types of memory resources, but in most if not all the cases, they can be divided into two major categories: *heap* and *non-heap* memory.

7.5.1 Heap

The Java virtual machine usually has a heap that is shared among all threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated. It is created at Java virtual machine start-up. Heap memory for objects is reclaimed by an automatic memory management system known as garbage collector.

The heap may be of a fixed size or may expand and shrink. The memory for the heap does not necessarily need to be contiguous.

7.5.2 Non-heap Memory

The Java virtual machine also manages memory resources other than the heap, these are usually referred as non-heap memory.

The usage of the non-heap memory is implementation specific and is usually associated with the internal processing of the JVM (JIT working buffers, JIT code buffers, etc). In addition, some JVM implementations might also store runtime constant pool, field and method data, and the code for methods and constructors in what is known as “method area” in the non-heap memory.

The method area is logically part of the heap but a JVM implementation may choose not to either garbage collect or compact it. Similar to the heap, the method area may be of a fixed size or may be expanded and shrunk. The memory for the method area does not necessarily need to be contiguous.

7.5.3 Common Abstract Model

Memory pools and memory managers are the abstract entities for monitoring and management of the JVM’s memory subsystem.

The JVM has at least one memory pool and it may create or remove memory pools during execution. A memory pool can be part of the heap or part of the non-heap memory.

The memory manager is responsible for managing one or more memory pools. The garbage collector is one type of memory manager responsible for reclaiming memory occupied by unreachable objects. The JVM may have one or more memory managers. It may add or remove memory managers during execution. A given memory pool can be managed by more than one memory manager.

The memory pool has five attributes:

- initSize

  Initial amount of memory requested by the JVM, from the operating system, for a given memory pool. The JVM may request additional memory from the operating system later when appropriate. Its value may be undefined.

- Used

  The amount of memory currently in use.

- Committed

The amount of memory that is guaranteed to be available for use by the JVM. The amount of committed memory may change over time (increase or decrease). It is guaranteed to be greater than or equal to initSize.

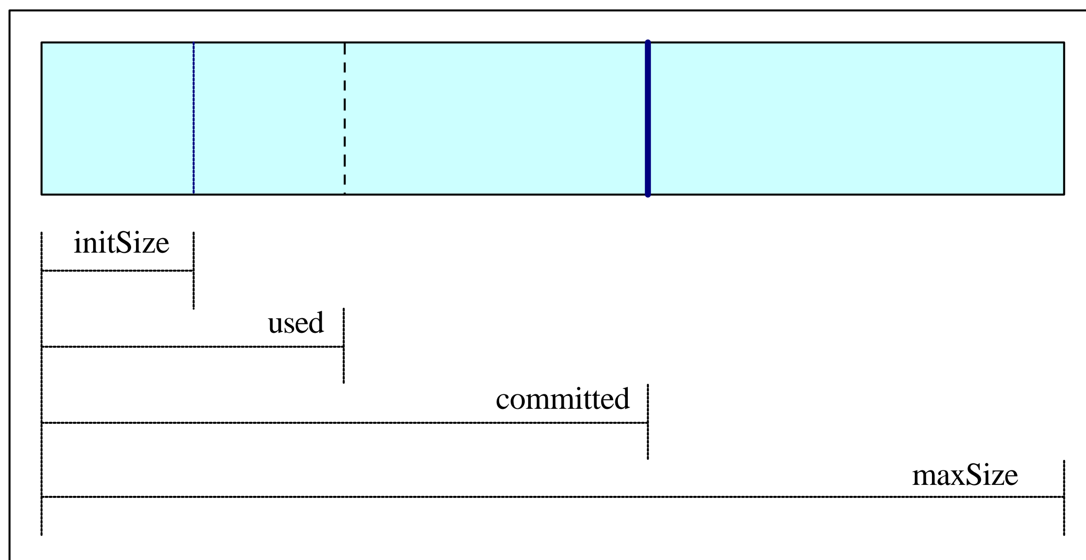
- maxUsed

  The maximum amount of memory ever used since the given memory pool was created.

- maxSize

  The maximum amount of memory that can be used for a given memory pool. The maximum amount of memory for memory management could be less than the amount of committed memory. Its value may be undefined.

This picture depicts a memory pool with maxSize > committed.



7.5.4 Examples

A Java virtual machine exposes three memory pools and two memory managers for monitoring and management:

- memory pool A for object allocation - managed by mark-sweep-compact collector
- memory pool B for method area - managed by mark-sweep-compact collector
- memory pool C for keep compiled native code used by JIT - managed by malloc/free manager

Memory pool A belongs to heap memory. Memory pool B and C belong to non-heap memory. Mark-sweep-compact collector is one memory manager managing pools A & B. Malloc/free memory manager manages pool C.

Mandatory	Optional

Memory Pool (list of memory pools which may vary over time)
 	
• Name (not necessarily unique)
• Membership
  o heap memory
  o non-heap memory
• Init size *(may be unspecified)*
• Used size
• Committed size
• Max size (may be unspecified)
• Max used size
• Alarm Threshold *(default is platform specific)*
• List of memory managers managing this memory pool
 	
Memory Manager (List of memory managers which may vary over time)
 	
• Name (not necessarily unique)
• List of memory pools managed by this memory manager
 	
Heap
 	
• initial amount of memory that the JVM allocates
• amount of used memory
• amount of committed memory *(memory guaranteed to be available for the JVM to use)*
• maximum amount of memory that the JVM will attempt to use
• Snapshot of the number of objects pending finalization (approx. value)
 	
Garbage Collector (one kind of Memory Manager)
 	
(some of this data may be undefined)
 	
• Current number of collections
• Total accumulated elapsed time spent in GC
 	
Manageable Attributes
 	
• Turn on/off -verbose:gc
• Set alarm threshold for a given memory pool
• System.gc
	Low memory detection support:
	
	• Heap Alarm
	
	The alarm will be triggered when the amount of available memory of any one of the memory pools of the heap is insufficient as per user defined threshold.
	
	• Non-heap Alarm
	
	The alarm will be triggered when the amount of available memory of any one of the memory pools of the non-heap is insufficient as per user defined threshold.

Table 7.5-1 Memory Subsystem

Part 2. Java Language Application Programming Interface

This section describes the requirements for the Java Language API for accessing the JVM monitoring and management data.

8 Requirements for the Java API

8.1 Production Environment

The interface will be used in the production environment.

8.2 Access to JVM Monitoring Data

The interface shall provide access to the JVM monitoring data defined in Chapter 7.

8.3 On-Demand Monitoring

The interface shall support on-demand monitoring to enable and disable costly data collection.

8.4 24x7 Operations, No Restart

No restart shall be required for enabling, disabling and resetting the data monitoring.

8.5 Multi-Thread Safe

The interface shall support use by multiple threads.

8.6 Optional Data/Operation

The interface shall provide a mechanism to query whether or not a given operation is supported by a JVM.

8.7 Extensible Mechanism

The interface shall allow a vendor to provide their platform-specific support through an extensible mechanism.

8.8 JMX MBeans

JMX MBeans shall be defined to enable remote JVM monitoring and management.

Part 3. Native Interface

This section describes the general characteristics of the native interface for externalizing the JVM monitoring and management data and for exporting management *knobs and buttons*. ***The requirements and characteristics listed in this section are additional to the general requirements describe in "Part 1" of this document.*** The minimum requirement is for native interface to provide the necessary APIs to externalize the same list of monitoring data and controls required in "Part 1" and/or to enable counters and accumulators to be computed by a native agent. JVM implementations may provide some of these data and controls by use of the self-describing extensions mechanism present in the JVM Tools Interface (JVMTI) currently being specified on JSR163. Both "Part 1" and "Part 3" provide guidelines for the interfaces to be implemented in the JVMTI.

9 General requirements for the Java native interface

9.1 Dynamic Load of Native Agents and Late Binding

The native interface shall support the late binding of native agents without restarting the JVM.

The JVM monitoring and management capabilities shall all be available after the JVM initializes.

9.2 Support for Multiple Concurrent Agents

The native interface shall support connection by multiple native agents and guaranteed correct coordination.

The native interface shall support multi-thread safe operation.

The native interface shall provide APIs to aid the coordination of multiple agents (e.g. raw monitors).

9.3 24x7 Operation without Restarting the JVM

The native interface shall support enable/disable/reset data collection without the need to restart the JVM.

9.4 Extensions capabilities

The native interface shall support a query mechanism for native agents to discover platform specific extensions supported by a given JVM.

9.5 Rules for writing agent code

Rules and best practice programming guidelines shall be provided to help end users and tools vendors wanting to exploit the JVM monitoring and management capabilities.

10 Additional requirements

10.1 Operating System, Runtime and JIT Compiler Properties

The minimum requirement is for the same list of system properties required in "Part 1 (Chapter 7.2)" to be available through the native interface. Ideally all the systems properties supported by a given JVM should be available.

Interface Requirement	JVMTI supported
Mandatory System Properties • List of "System Properties" keys supported on a given JVM • System Property value for a given key	**YES** • GetSystemProperties • GetSystemProperty
Optional Boot class path	**No**

Table 10.1-1 Native Interface - OS, Runtime and JIT properties

10.2 Execution and Synchronization Subsystem

Thread lifecycle in conjunction with object locking (a.k.a. monitor lock) data can be a powerful tool for identifying that contention is taking place and also to derive thread dependency (thread graph per monitor).

Thread lifecycle can be derived from thread start/blocked/end events and monitor lock data shall be obtained through a query mechanism. This functionality is currently provided by the JVMTI spec.

10.3 Class Loading Subsystem

The native interface shall implement the functionality necessary for the monitoring of lifecycle of classes, class identification and classloader statistics.

Data concerning the class loading subsystem can be grouped on a *classloader basis* and/or *class basis*. Lifecycle shall be derived from data

surfaced through a notification mechanism. Statistics and aggregations should be computed by the agent using data collected through query. Query only happens as result of a request therefore any overhead can be at least managed. This functionality is currently provided by the JVMTI spec.

10.4 Memory Subsystem

Currently, the JVMTI support for accessing monitoring data from the memory subsystem is not very flexible and assumes a given implementation. As the *memory manager* technology continues to evolve and as different JVM vendors can potentially implement _very_ different memory models and memory managers, our recommendation is for vendors provide their own metric through the JVMTI extensions mechanism.

Part 4. Appendixes

Appendix A. Expert Group Collaboration Agreement

JSR 174 Expert Group Participation Rules

Purpose:
The purpose of JSR 174 is to develop the Monitoring and Management Requirements Specification for the Java™ Virtual Machine. To do that, the members of the expert group will have to exchange information, ideas, documents, code, etc. Sun has represented to IBM that it has agreements with each of you, or with the companies that you represent, to enable your participation in this expert group. According to Sun, these agreements provide licenses to any contributions that are made sufficient for IBM to produce the Requirements Specification for this JSR and license it under the proposed terms.

General collaboration terms:
Due to the nature of this expert group, no information or material which is considered confidential to you or your employer should be submitted as a contribution. All information or material provided to this group will be considered non-confidential.

Licensing Terms:
This draft specification is not final. Any final specification that may be published will likely contain differences, some of which may be substantial. Publication of this draft specification is not intended to provide the basis for implementations of the specification. This draft specification is provided AS IS, with all faults. THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF CONDITION OF TITLE OR NON-INFRINGEMENT. You may copy and display this draft specification provided that you include this notice and any existing copyright notice. Except for the limited copyright license granted above, there are no other licenses granted to any intellectual property owned or controlled by any of the authors or developers of this material. No other rights are granted by implication, stopple or otherwise.

Appendix B. Trademarks

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.