

Oracle JRockit Mission Control Overview

An Oracle White Paper
June 2008



Oracle JRockit Mission Control Overview

Oracle JRockit Mission Control Overview.....	3
Introduction.....	3
Non-intrusive profiling and diagnostics.....	3
Management Console.....	4
JRA (JRockit Runtime Analyzer).....	8
Memory Leak Detector.....	12
Conclusion.....	15

ORACLE JROCKIT MISSION CONTROL OVERVIEW

JRockit Mission Control provides non-intrusive profiling and diagnostics of JVMs, making it suitable for production time use.

INTRODUCTION

Oracle JRockit Mission Control is a set of powerful tools running on the Oracle JRockit JVM. These tools deliver advanced, unobtrusive JVM monitoring and management, suitable for use both in development and production environments. This article gives an introduction to JRockit Mission Control, describing the main components in the suite, how this suite's components differ from competing technologies, and how you can use them to monitor, manage, profile and diagnose your applications when running on the JRockit JVM.

NON-INTRUSIVE PROFILING AND DIAGNOSTICS

Most technologies used today to monitor, manage, and profile the Java runtime use fairly intrusive technologies, like byte code instrumentation and **JVMTI** (which has replaced the older **JVMPi**). The main focus of JRockit Mission Control is to gather the data necessary with the lowest possible impact on the running system. The technology used also enables the application to run at full speed once the tool is disconnected from the JVM. This makes JRockit Mission Control suitable for use in production environments. The minimal overhead also minimizes the Heisenberg effect and can provide more representative data for your application than the more overhead-prone techniques.

JRockit Mission Control currently contains three main tools:

- **Console**
A JMX based management console
- **JRA**
The main profiler which works like a flight recorder
- **Memleak**
A tool for detecting and hunting down memory leaks

JRockit Mission Control currently contains three powerful tools:

The Management Console

The JRockit Management Console is a tool for monitoring and managing multiple JRockit instances. It captures and presents live data about GC pauses, memory and CPU usage, as well as information from any JMX MBean deployed in the JVM's internal MBean server. JVM management includes dynamic control over CPU affinity, garbage collection strategy, memory pool sizes and more.

The JRockit Runtime Analyzer

The JRockit Runtime Analyzer (JRA) is an on-demand 'flight recorder' that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA Mission Control plugin. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, object statistics, and latency events.

The Memory Leak Detector

This is a tool for discovering, and finding the cause for, memory leaks. The JRockit Memory Leak Detector's trend analyzer can discover very slow leaks, it shows detailed heap statistics including referring types and instances to leaking objects, allocation sites, and provides quick drill down to the cause of the leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand the sometimes complex information.

The next three chapters will explain each of these tools in a little bit more detail.

Management Console

The Management Console is a JMX console you can use to monitor any JMX compliant JVM.

To fully utilize the capabilities of the Management Console you need to be monitoring a JRockit JVM.

The Management Console is a JMX-based console for managing and monitoring the JRockit JVM. It provides vital health data like the live set, the heap usage, CPU load, and other attributes exposed by the MBeans registered in the JVM-internal platform MBean server. The Management Console also includes a low overhead on-line method profiler and an exception counter.

The Management Console consists of an agent running in the JRockit process, exposing the MBeans registered in the JVM-internal platform MBean server, and a separate GUI plug-in running in JRockit Mission Control. When the Management Console connects to a JRockit JVM, a set of JRockit specific MBeans will automatically be created and registered in the platform MBean server, exposing JRockit specific functionality.

As **Figure 1** illustrates, a single Management Console can be connected to several JRockit JVMs, and several instances of the Management Console can be connected to a single JRockit JVM. Note that since JRockit Mission Control can handle multiple JRockit JVMs, there is usually no need to run multiple instances of JRockit Mission Control on the same machine.

You can connect several management consoles to different JVMs from one JRockit Mission Control.

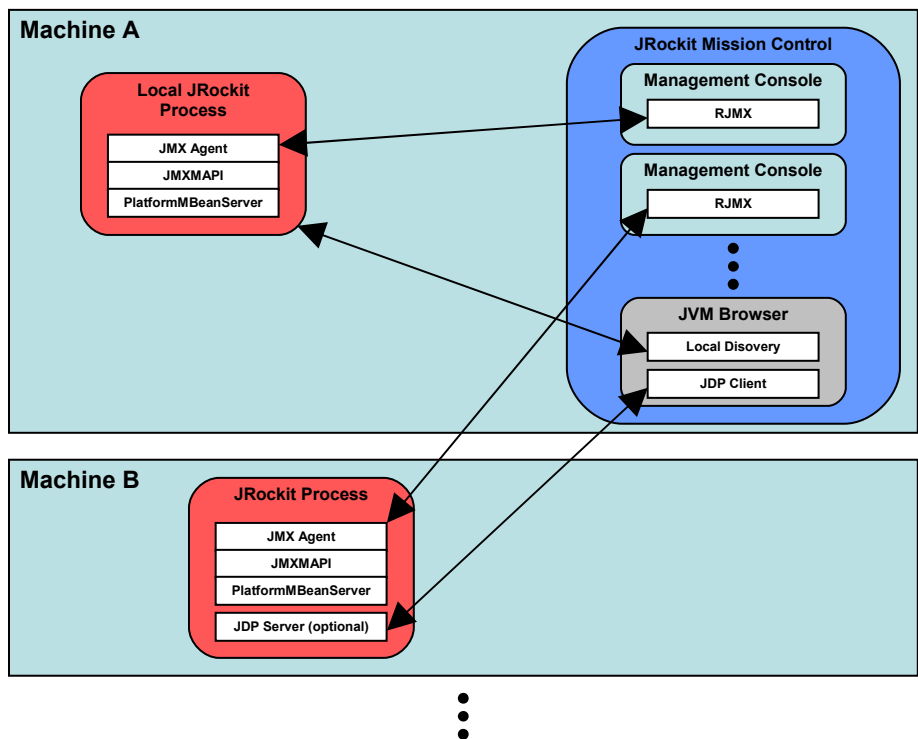


Figure 1: BEA JRockit Management Console communication

From a high-level architectural perspective, the console relevant parts of the monitored JRockit includes:

The JRockit JVM contains proprietary MBeans exposing JRockit specific data to the Management Console, an agent that allows the console to access the data remotely and a server that enables automatic discovery of the running JRockit.

- **A set of interfaces (JMXMPI)**
JRockit specific MBeans that at a very low overhead delivers JRockit specific data. The collection of MBeans includes a dynamically generated MBean exposing all of JRockit's performance counters as attributes. These extensions are internally named the JRockit JMX Management API, or JMXMPI for short.
- **An agent exposing JMXMPI**
Communication with the console uses remote JMX over RMI. You can start the agent using JRockit JVM command line parameters. The agent life cycle can also be controlled by various JRockit specific APIs and jrcmd, which is a command line utility for the JRockit JVM provided with the JRockit JDK.
- **The JDP (JRockit Discovery Protocol) Server**
The Management Console carries out automatic detection of JRockit JVMs running on the network by using multicasting to receive the location of the particular JRockit. The JDP Server is optional, and can be started using JRockit JVM command line parameters or jrcmd.

Note that in all versions of JRockit Mission Control other than the 1.0 version, JDP is an extension to the JVM Browser, and not part of the Management Console.

The console also sports an on-line, exact, method profiler that provides invocation counts and method timing information. The recommended way to do profiling in JRockit Mission Control is however to use the JRockit Runtime Analyzer (JRA).

The console client plug-ins sports a JMX service layer that provides persistence, attribute subscriptions and notifications, automatic JVM discovery and an extensible framework to write new plug-ins for the management console.

From a high-level architectural perspective, the Management Console client side plug-ins includes:

- RJMX (JRockit Remote JMX services)**
 Provides services such as persistence, the attribute subscription abstraction, and the notification framework.
- The JDP Client**
 Automatically discovers JRockit JVMs that have JDP enabled. *(Again, this is only valid for the first version of Mission Control. In more recent versions the JDP Client is an extension to the JVM browser.)*
- The main console plug-in**
 Defines the framework for the Management Console and the extension points for other plug-ins extending the Management Console.
- Various plug-ins**
 A set of plug-ins that make out the default set of tabs that you can see in the Management Console, such as the Overview, MBean Browser and Method Profiler.

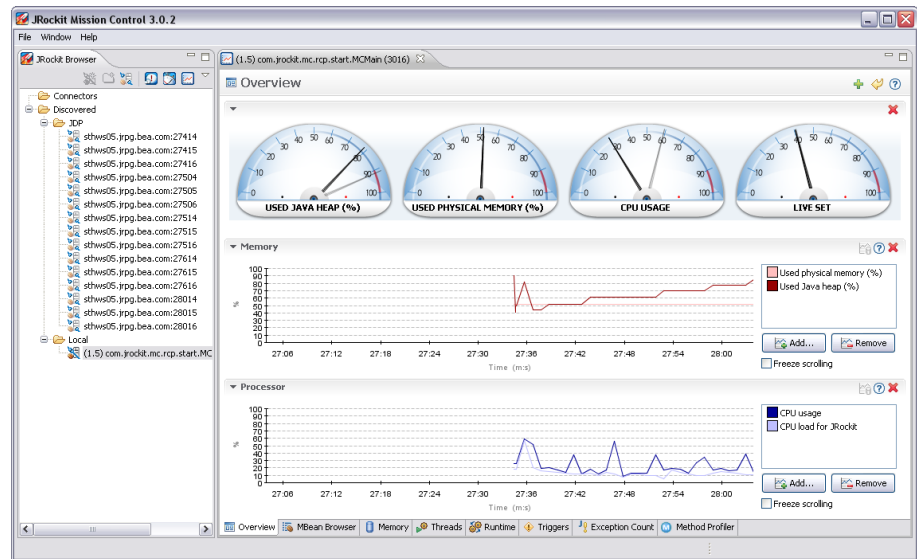


Figure 2: The Management Console GUI

The Management Console application introduces the concept of an Attribute Subscription, which, somewhat simplified, is defined by the MBean ObjectName, the Attribute Name, and the subscription interval. The console allows you to add notification rules, plot and persist data from such attribute subscriptions. The attribute subscriptions can be based on regular JMX MBean attributes, on individual composite data keys from an attribute, or on JMX notification data. You can create your own composite attribute subscriptions that depend on several other attribute subscriptions for its data, or even create synthetic attribute subscriptions

where it's up to you, the implementor, to provide the data by whichever means you would like.

The Method Profiler provides a very low-overhead means of finding out how many times a method is being invoked, and how much time is being spent in that particular method. The machine code of the methods of interest are simply regenerated with a tiny amount of instrumentation code, which is removed the instant the profiler is stopped. The overhead of using the method profiler therefore is only incurred when you have selected methods for profiling, and only for the selected methods.

This kind of method profiling is nevertheless only useful when you already know in advance what you're interested in looking at - in most cases you should use the JRockit Runtime Analyzer for profiling.

To summarize, the Management Console is a very flexible JMX monitor and management tool that provides a wealth of features:

- Plotting of any numerical attribute
- Persistence of any set of attributes for offline analysis
- Special attribute subscriptions for easy plotting of pausetimes, live set size, and continuous heap usage
- Notification rules that can take actions when a user-specified condition occurs for a certain attribute
- The ability for users to plug in their own code for both notification actions and constraints
- Management of the Java runtime, including dynamically changing the garbage collection strategy, heap size, nursery size, JRockit process affinity, enabling/disabling of verbosity flags, and more
- A low overhead method profiler
- An exception counter
- A means to dynamically invoke the operations of your MBeans
- A way to invoke the JRockit diagnostic commands

For more information on how to configure the console to use SSL, authentication and roles, see the [JRockit documentation](#).

The JRockit Runtime Analyzer was originally built to provide feedback to the JRockit developers. It is used both internally in the JRockit support organization and by customers for diagnosis, profiling and tuning of both the JVM and Java applications running in the JVM.

JRA (JRockit Runtime Analyzer)

JRA is a Java application and JVM profiler. It has been around for quite some time within the JRockit development team, and was originally created to let the JRockit developers find good ways to optimize the JVM based on representative data from real-world applications. It has since proven very useful to customers for solving problems in both production and development. JRA is also one of the primary tools used by the JRockit support organization to help resolve customer issues.

JRA works like a flight recorder; it records how the Java application and the JVM behave for a preset period of time. You can then analyze the recording using the JRA Mission Control plug-in, where, for example, call traces for hot methods, bad synchronization, and other important application/JVM behavior can be analyzed. The Oracle Support organization uses JRA recordings from customers on a regular basis to help Oracle customers resolve issues.

JRA consists of two parts: the recording engine in the JVM, and the set of GUI plug-ins you can use to analyze the resulting recording. The recording engine uses several sources of information including the JRockit Hot Spot Detector (also used by the optimization engine to decide what methods to optimize), the operating system, the memory management system (most notably the garbage collector), and the JRockit lock profiler, if enabled.

With the JRA tool you can easily analyze the information contained in the JRA recording by graphical means (see **Figure 3**). Graphs show pause times, heap usage, and the committed heap size during the recording period. You can select any individual garbage collection (GC) to view very detailed information about that particular garbage collection.

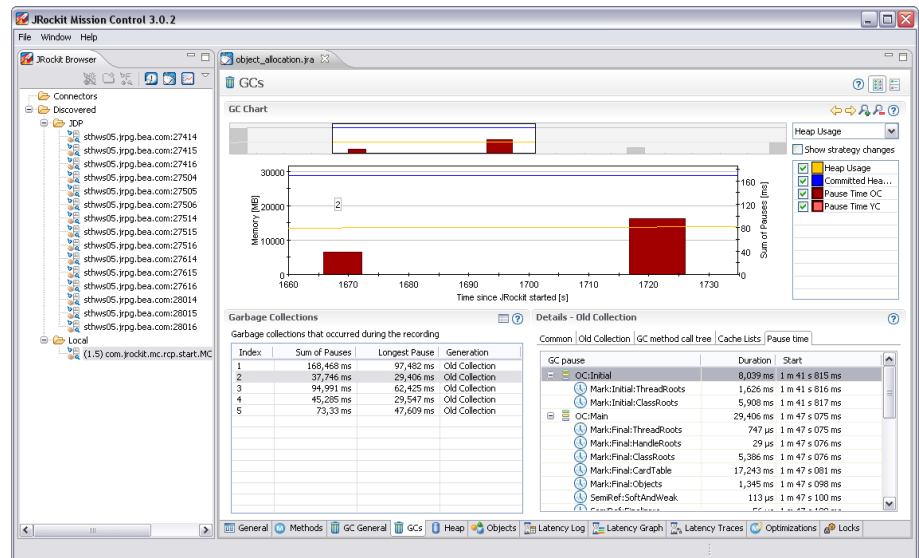


Figure 3: JRA Garbage Collection Information

The JRockit Runtime Analyzer provides detailed garbage collection profiling, method profiling, latency events, lock profiling, heap histograms and more.

You get call traces for the hot methods—not only the call traces leading up to the call of the method, but also the call traces portraying what was usually called next. The call traces also show whether or not an optimized version of the method was called.

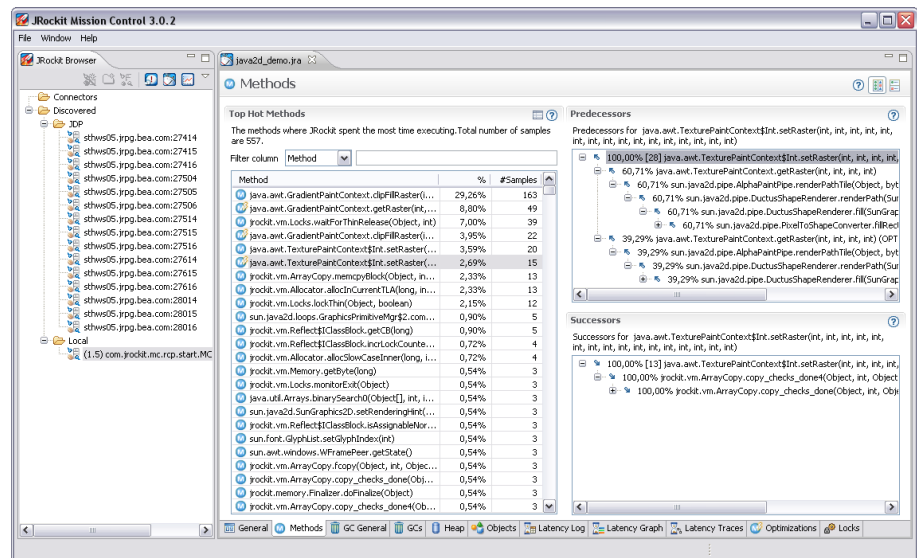


Figure 4: JRA Method Profiler

A heap histogram is taken at the beginning and at the end of the recording, revealing the heap usage per class, for instances of classes occupying more than 0.5 percent of the used heap space. The collected heap information is also used to render pie charts showing heap utilization.

Since JRockit R27.3 Mission Control contains a **latency analyzer** able to profile and graphically visualize latencies in Java applications. Mission Control has always been good at pinpointing where performance is suffering due to pure computational overhead, i.e. CPU bound problems. Before R27.3, however, there was really no good way to find the cause of latency related problems, i.e. problems when throughput in your application suffers since threads are stalling for different reasons.

With R27.3 and the latency detection tool you can see thread graphs of where the threads are stalling. Thanks to powerful and well optimized visualization techniques it is possible to visualize, zoom and pan among hundreds of thousands of events very quickly. You can also look at a method call tree to see from where the events are occurring, or view the raw events in a log view with assorted filtering features.

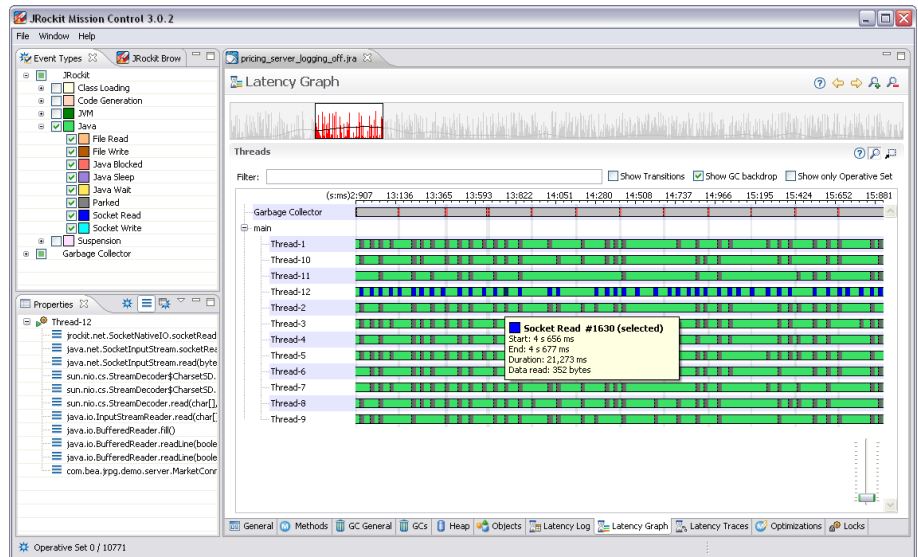


Figure 5: JRA Latency Tool

There are several ways to start a JRA recording:

- Use the JRockit Mission Control GUI.
- Use the JRCMD – a command line tool for the JRockit JVM
- Use the JRockit extension MBeans.
- Use the JRA -XXjra command-line parameters.

You can also trigger JRA recordings from the Management Console using Notification Rules; for example, a rule can be created to start a JRA recording when the CPU load has been above 90 percent for a minute.

To sum things up, the JRA is a powerful JVM and Java application profiler that provides:

- Efficiency (typically less than 2 percent overhead) in profiling both the JVM and Java application.
- Method call traces showing which paths were taken to get to the method, and what was called next.
- Method hot spot table, showing which methods were invoked most often.
- Very detailed garbage collection statistics, showing what happened during each individual GC.
- Garbage collection strategy changes.
- Graphical plots of heap usage and pausetimes.
- Heap histograms showing the heap usage per class at the beginning and the end of the recording.
- Detailed lock profiling, showing which locks were taken, if they were contended, how many times they were taken, and more.
- Pie charts of heap usage per object size, including fragmentation.
- Which methods were optimized during the recording.
- Latency profiling

For more information about using the JRA, see the [JRockit Mission Control documentation](#).

Memory Leak Detector

The Memory Leak Detector quickly spots even slow memory leaks. It does the analysis on-line, while the application is running at full speed.

The Memory Leak Detector is a tool to help you quickly discover memory leaks. Even though the automatic memory management of Java frees the developers of the burden to explicitly allocate and free memory used, memory leaks can still occur if the program unwittingly keeps referencing objects that are no longer of any use.

The Memory Leak Detector gives you a trend analysis to quickly spot even slow memory leaks. The trend analysis shows the heap usage of your application per class. It will tell you how much space instances of the type are using, what fraction of the heap they are occupying, how many instances there are, and how quickly that usage is growing in bytes per second.

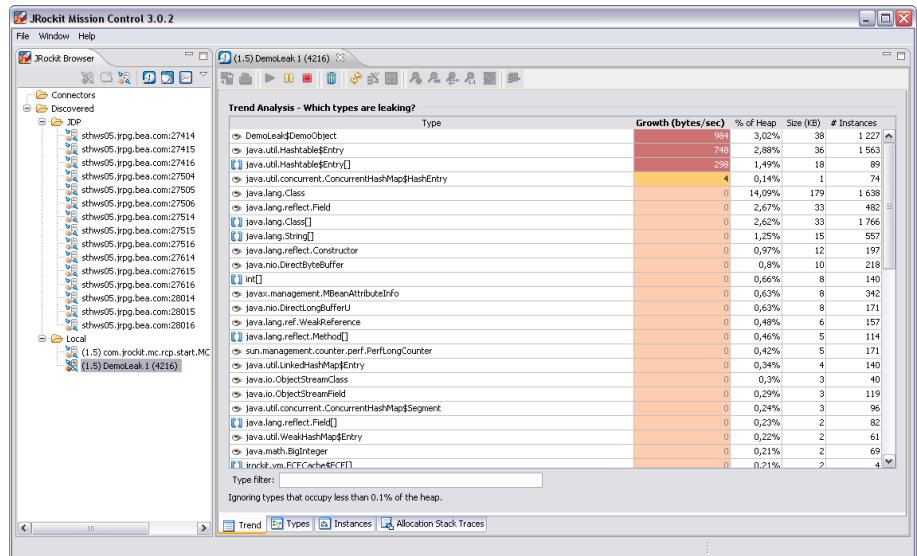


Figure 6: Memleak Trend Table

The JRockit Memory Leak Detection Tool can show relationships between different types on the heap as well as instances in interactive graphs.

The Memory Leak Detector also provides the means to quickly drill down to the cause of the leak. You can select a suspected type in the trend analysis table, and have the types with instances pointing to the selected type shown in a graph (see Figure 7). The nodes of the graph can be arbitrarily expanded to let the user back-track to what is ultimately causing the references to be held. Instances of a class can be shown and introspected, and all instances pointing to a selected instance can be shown in an instance graph. You can also turn on allocation traces to track all allocations of instances of a specific class.

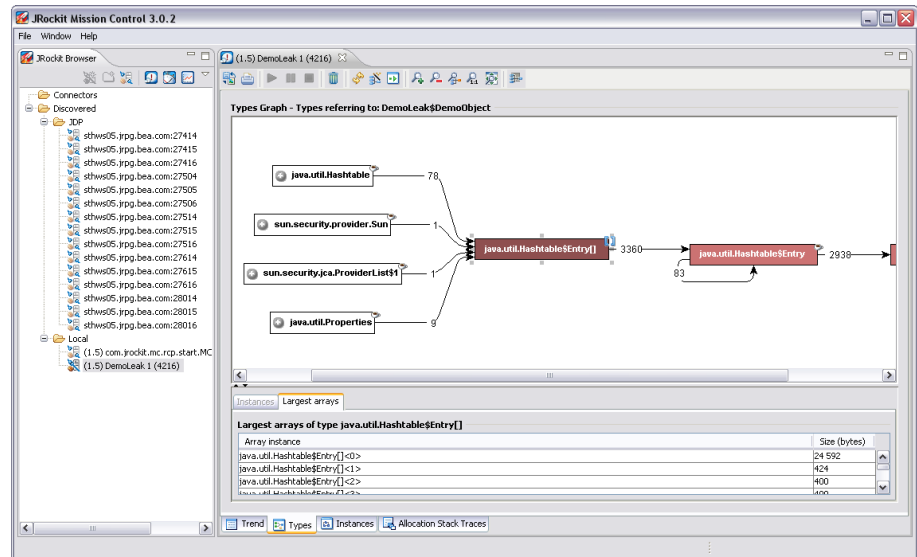


Figure 7: Memleak Class Graph

When a Memory Leak Detector session is started, a Memory Leak Server (MLS) starts. Depending on the JDK version of the JRockit JVM you connect to, either the JMX connector in JLMEXT (5.0), or the Rockit Management Protocol (RMP) (1.4) will be used. Note that the protocols differ and that there are differences in networking characteristics and security. A Memory Leak Server is a native server with which the rest of the communication during the session takes place. On the client side a Java API is used to communicate with the native server (see Figure 5). The reason for using a native server is that if a serious memory leak condition occurs and the JRockit JVM runs out of Java heap, JRockit will no longer be able to run Java code.



Figure 8: Establishing a Memleak Session

The Memory Leak Detection Tool can achieve the performance it does by piggybacking on the mark phase of the garbage collector, adding a little bit of extra bookkeeping whilst doing what it has to do anyway – collect garbage.

The Memory Leak Detector piggybacks on the mark phase of the garbage collector, adding some book-keeping to record the heap histograms (the memory statistics of the heap aggregated per class) and generate the trend analysis. One common solution to finding memory leaks in competing tools is to take several snapshots of the entire heap of an entire system, and then compare the snapshots. In a system with hundreds of gigabytes of heap or a system in a production environment, this approach may not be feasible. With the JRockit Memory Leak Detector, only the information you are interested in is sent over the wire, making the bandwidth demands very small. When you use the allocation call trace functionality, only code involving allocation sites for the selected class is instrumented. Also, as soon as the session is over, or the allocation traces are turned off, the instrumentation is removed and the code involving these allocation sites returns to full speed.

To summarize, the JRockit Memory Leak Detector is an advanced analysis tool with a number of novel features:

- It can perform a trend analysis that discovers even slow memory leaks.
- It does the analysis online, piggybacking on the JRockit Memory Manager, instead of, for instance, dumping the entire heap to the client multiple times and examining the difference.
- It provides an advanced user interface that lets you find and examine the relationships between the leaking type and other types, or the leaking instance and other instances.
- The detector has a very low overhead and can be used to do the analysis online, in production.
- No byte code instrumentation is used; when analyzed with the Memory Leak Detector, the Java code will continue executing as if it was never connected. No Java code will be modified.

For more information about the Memory Leak Detector, see the JRockit Mission Control [documentation](#).

CONCLUSION

JRokit Mission Control is a versatile tools suite for monitoring, managing, profiling, and eliminating memory leaks in your Java applications. JRokit Mission Control is free for development use. You can reliably use JRokit Mission Control in production environments without leaving any trace in your system after it has been used, and with a much smaller performance overhead than comparable tools when it is in use.



JRockit Mission Control Overview

June 2008

Author: Marcus Hirt

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2008, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.

Other names may be trademarks of their respective owners.