BEA WebLogic® Real Time

Introduction to WebLogic Real Time

Version 2.0
July 2007

Contents

CHAPTER 1

Overview of WebLogic Real Time 2.0

This section contains information on the following subjects:

What is WebLogic Real Time?

WebLogic Real Time provides lightweight, front-office infrastructure for low latency, event-driven applications. For companies in highly-competitive environments where performance is key and every millisecond counts, WebLogic Real Time provides the first Java-based real-time computing infrastructure.

For example, for certain types of applications, particularly in the Telecom and Finance industries, stringent requirements are placed on transaction latency. When these applications are written in Java, the unpredictable pause times caused by garbage collection can have a profound and potentially harmful affect on this latency.

For this reason, WebLogic Real Time's proprietary BEA JRockit R27.3 JVM features *deterministic garbage collection*, a dynamic garbage collection priority that ensures extremely

short pause times and limits the total number of those pauses within a prescribed window. Such short pauses can greatly lessen the lessen the impact of the deterministic garbage collection when compared to running a normal garbage collection.

WebLogic Real Time supports Java applications running in the following environments:

- WebLogic Event Server 2.0
- WebLogic Server 10.0 and later
- WebLogic Server 8.1 (SP2 and later)

For a complete listing of supported configurations with WebLogic Server releases, see "Supported Configurations for WebLogic Real Time" on page 1-8.

WebLogic Real Time also supports standalone Java applications running on J2SE 1.4.2 and 5.0 runtime environments, as well as Spring Framework-based applications, as described in "Software Components" on page 1-5.

New JRockit Features in WebLogic Real Time 2.0

WebLogic Real Time 2.0 is bundled with BEA JRockit(R) JDK 5.0 R27.3.0. This version of JRockit includes the full version of Mission Control, which is a suite of tools designed to monitor, manage, profile, and gain insight into problems occurring in your Java application without requiring the performance overhead normally associated with these types of tools.

Mission Control includes the following two tools that are of particular interest to WebLogic Real Time 2.0 users:

- JRockit Latency Analyzer Tool
- JRockit Memory Leak Detector

JRockit Latency Analyzer Tool

The Latency Analyzer Tool, part of the JRockit Runtime Analyzer (JRA) helps you work your way down to a Java application latency. You can use the **Latency Graph** to visually see how a Java application that contains latencies looks like. This tool gives you great flexibility to pinpoint where in the code waits and other latencies occur.

To record latency data, you need to create a JRA recording. Before you start the JRA recording, you must select one of the Latency Recording profiles in order to record latency data.

See BEA JRockit Runtime Analyzer for additional information about using the latency analyzer tool and JRA recordings to record latency data. After you launch JRockit Mission Control, you can also access additional documentation about this feature using online help.

JRockit Memory Leak Detector

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

See Introduction to JRockit Memory Leak Detector for additional information about using the memory leak detector. After you launch JRockit Mission Control, you can also access additional documentation about this feature using online help.

Starting JRockit Mission Control

To start JRockit Mission Control, follow these steps:

1. Ensure that your `JAVA_HOME` environment variable points to the root folder of the JRockit JDK included in WebLogic Real Time 2.0.

   This directory is called *`BEA_HOME`*`\jrockit-realtime20_150_11`, where *`BEA_HOME`* refers to the main BEA Home directory into which you installed WebLogic Real Time 2.0, such as `d:\beahome_wlrt`.

2. Open up a command window.

3. Execute the `jrmc` executable file, located in the `%JAVA_HOME%\bin` directory:

   ```
   (Windows)  prompt> %JAVA_HOME%\bin\jrmc
   (Linux)    prompt> ${JAVA_HOME}/bin/jrmc
   ```

Example Use Cases

These use cases provide examples of how WebLogic Real Time can provide solutions for high-performance environments with response-time sensitive applications.

Derivative Exchange Defies Arbitrage Traders

An investment arm of a large retail bank provides an exchange for derivatives of European securities. It is an over-the-counter (OTC) request-for-quote and execution system (but provides no settlement and clearing services). A broker submits a request for a quotation and includes the investment identifier and quantity. The system accepts the quotation and applies certain business rules. Depending upon the investment identifier and market conditions, the request is routed to a particular third-party market-maker who then calculates and provides the bid and ask price for the derivative. The response is returned to the broker via the OTC exchange. The broker can then execute the trade of the derivative through a subsequent request, which is routed via the OTC exchange to the appropriate market maker.

The complication with this arrangement is that arbitrage traders can take advantage of the latency delay in the bank's OTC exchange infrastructure because the arbitrage trader can measure the latency that occurs during the small period in which the request for quotation is handled. In a fast moving market, price changes of the derivative may occur within this latency period. This presents an opportunity for an arbitrage trader to take advantage of inefficiency in the marketplace and expose the investment bank to intolerable risk.

The investment bank requires a very high performance-driven software infrastructure, such as WebLogic Real Time. It requires that the latency of the OTC exchange be extremely low. Specifically, to combat arbitrage traders, the latency of the exchange's infrastructure must be less than the latency of the arbitrage traders' infrastructure. In this way, the arbitrage traders' data becomes stale before the exchange's, and therefore is not actionable.

Competition-Beating Risk Calculation Infrastructure

A large investment bank is a market-maker for fixed income securities. A request-for-quote (RFQ) is received from an inter-dealer market electronic communication network (ECN), such as TradeWeb. This RFQ would have been submitted to a number of entities. To be competitive, it is vital that the quotation is returned as quickly as possible with the best possible price. Therefore, a minimum amount of latency is necessary to ensure that the investment bank wins customers, or at least, the latency is less than that of the organization's competitors.

During the quotation process, a risk and pricing model is executed to determine the quote price to provide to the customer. Because of the complexity of these calculations, they are currently performed overnight. The result is a stratum of at least four grades of risk advisories that govern fixed rate securities prices. Note that there is at least a twelve-hour lag in these risk calculations. This leads to a risk window since the calculations are stale even at the start of next-day business. To lower this risk, and potentially provide better rates to customers, a real-time risk and pricing

calculator would be required. WebLogic Real Time provides a latency-adverse infrastructure to make this feasible.

Software Components

WebLogic Real Time supports Java applications running on WebLogic Event Server 2.0, WebLogic Server 10.0 (or higher), and WebLogic Server 8.1 environments, as well as supporting standalone Java applications running on J2SE 5.0 and 1.4.2 runtime environments.

WebLogic Real Time includes the following software components:

BEA JRockit 5.0 R27.3 JVM

The BEA JRockit® 5.0 R27.3 JVM is certified to be compatible with J2SE 5.0 (update 6), WebLogic Event Server 2.0, and WebLogic Server 10.0 or higher. The 5.0 R27.3 JVM includes the Deterministic Garbage Collector for dynamic garbage collection priority that ensures extremely short pause times and limits the total number of those pauses within a prescribed window, as described in "Deterministic Garbage Collection" on page 1-6. It also installs the JRockit Runtime Analyzer (JRA) tool, latency analyzer tool (LAT), and memory leak detector, which provide internal metrics that are useful for profiling JRockit, as described in "JRockit Runtime Analyzer (JRA)" on page 1-7.

The `realtime20_jdk1.5.0_`*`XXX`* version of the installer kit can be installed in a WebLogic Event Server 2.0 or WebLogic Server 10.0 environment to work with WebLogic Event Server or WebLogic Server applications, as well as in standalone mode for standalone Java applications or Spring Framework-based applications, with the following installation differences:

- **WebLogic Server 10.0 (and later) install mode** — The installer includes a WebLogic domain configuration template (`wl-realtime.jar`) for creating a 10.0 or later domain with Deterministic GC enabled. The installer also includes sample startup scripts, `startRealTime (.cmd/.sh)`, that demonstrate how to start BEA JRockit with Deterministic GC enabled, which may be useful for Spring-based applications that are using some WebLogic Server facilities.
- **Standalone install mode** — When installed in a BEA Home directory without WebLogic Server, the installer still includes the sample Deterministic GC startup scripts, which may be useful for Spring-based applications that are using some WebLogic Server facilities. However, in this scenario, the WebLogic domain template is not installed.

For a listing of the hardware and software configurations supported by WebLogic Real Time, see "Supported Configurations for WebLogic Real Time" on page 1-8.

BEA JRockit 1.4.2 R27.3 JVM

The BEA JRockit® 1.4.2 R27.3 JVM is certified to be compatible with J2SE 1.4.2_14 and WebLogic Server 8.1 SP2 and later. The 1.4.2 R27.3 JVM includes the Deterministic Garbage Collector for dynamic garbage collection priority that ensures extremely short pause times and limits the total number of those pauses within a prescribed window, as described in "Deterministic Garbage Collection" on page 1-6. It also installs the BEA JRockit Runtime Analyzer (JRA) tool, which provides internal metrics for Java developers using BEA JRockit as their runtime JVM, as described in "JRockit Runtime Analyzer (JRA)" on page 1-7.

The `realtime20_jdk1.4.2_XXX` version of the installer kit can be installed in a WebLogic Server 8.1 environment to work with WebLogic Server 8.1 SP2 and later applications, as well as in standalone mode for standalone Java applications or Spring Framework-based applications, with the following installation differences:

- **WebLogic Server 8.1 (SP2 and later) install mode** — The installer includes a WebLogic domain configuration template (`wl-realtime.jar`) for creating an 8.1 SP2 and later domain with Deterministic GC enabled. The installer also includes sample startup scripts, `startRealTime (.cmd/.sh)`, that demonstrate how to start BEA JRockit with Deterministic GC enabled, which may be useful for Spring-based applications that are using some WebLogic Server facilities.
- **Standalone install mode** — When installed in a BEA Home directory without WebLogic Server, the installer still includes the sample startup scripts, `startRealTime (.cmd/.sh)`, that demonstrate how to start BEA JRockit with Deterministic GC enabled. However, in standalone mode, the WebLogic domain template is not installed.

For a listing of the hardware and software configurations supported by WebLogic Real Time, see "Supported Configurations for WebLogic Real Time" on page 1-8.

Deterministic Garbage Collection

Memory management relies on effective *garbage collection*, which is the process of clearing dead objects from the heap, thus releasing that space for new objects. WebLogic Real Time uses a dynamic "deterministic" garbage collection priority (`-Xgcprio:deterministic`) that is optimized to ensure extremely short pause times and limit the total number of those pauses within a prescribed window.

For certain types of applications, particularly in the Telecom and Finance industries, stringent requirements are placed on transaction latency. When these applications are written in Java, the

unpredictable pause times caused by garbage collection can have a profound and potentially harmful affect on this latency.

However, shorter deterministic pause times do not necessarily equal higher throughput. Instead the goal of the deterministic garbage collection is to lower the *maximum* latency for applications that are running when garbage collection occurs. Such shorter pause times should lessen the impact of the deterministic garbage collection compared to running a normal garbage collection.

For more information on the deterministic garbage collector, see the BEA JRockit Diagnostics Guide.

Enabling the Deterministic Garbage Collector

- For standalone or Spring-Based Java applications, either:
  - Enter the `-Xgcprio:deterministic` option from a Java command line.
  - Use the sample startup scripts, `startRealTime (.cmd/.sh)`, that demonstrate how to start BEA JRockit with Deterministic GC enabled.
- For WebLogic Server environments, either:
  - Start WebLogic Server with the `-Xgcprio:deterministic` option in the startup script.
  - Use the Start menu > BEA Products > Tools > Configuration Wizard to create a domain with Deterministic GC enabled.

JRockit Runtime Analyzer (JRA)

The JRockit Runtime Analyzer (JRA) tool is an application that helps you profile your application and the Java runtime. It provides a wealth of useful metrics that are useful when using BEA JRockit as your runtime VM.

The BEA JRockit Runtime Analyzer consists of two parts. One is running inside the JVM and recording information about the JVM and the Java application currently running. This information is saved to a file which is then opened in the other part: the analyzer tool. This is a regular Java application used to visualize the information contained in the JRA recording file.

The JRocking Runtime Analyzer is packaged as part of the BEA JRockit Mission Control 2.0 tools suite. Documentation for Mission Control 2.0 is bundled with the tools as online documentation. For general information about Mission Control 2.0, see Introduction to BEA JRockit Mission Control.

Supported Configurations for WebLogic Real Time

For information on supported configurations, see BEA WebLogic Real Time 2.0 in *Supported Configurations: WebLogic*.

Terminology

Table 1-1 defines the terms and acronyms used this document:

Table 1-1 Terminology

Terms	Definition
Real-time	A level of computer responsiveness that a user senses as sufficiently immediate or that enables the computer to keep up with some external process (for example, to present visualizations of the weather as it constantly changes).
Latency	An expression of how much time it takes for data to get from one designated point to another.
Throughput	The amount of work that a computer can do in a given time period.
Deterministic garbage collection	Short, predictable pause times for memory heap garbage collection, which is the process of clearing dead objects from the heap, thus releasing that space for new objects.

CHAPTER 2

Tuning Real Time Applications for Deterministic Garbage Collection

This section contains the following guidelines for tuning your applications for the JRockit deterministic garbage collector that is included with WebLogic Real Time.

Note: For more information on adjusting other non-standard start-up commands available with JRockit, see the JRockit Configuration and Tuning Guide.

Basic Environment Tuning

Use these guidelines for configuring your environment to use WebLogic Real Time.

- *Ensure that CPUs are not at maximum capacity out on servers or clients*
  If an application takes a majority of the CPU, then the deterministic GC performance may actually degrade the average latency. The reason is that deterministic GC will do continuous GC and the GC will be competing with the application for CPU cycles. It is best that the CPU is not fully utilized to get the best latency. A best practice is to run your benchmarks at various loads (with and without deterministic GC) to determine the optimal load.

- *Too many active threads can cause increased latency due to context switching*
  The "sweet-spot" number is generally one thread per virtual CPU (i.e., counting dual-core and HyperTransport as separate CPUs), but leaving one CPU free for background GC work. However, if you make external calls (e.g., to a database), then it does make sense to allocating a few extra threads to utilize idle cycles.

  For information on tuning JRockit GC threads, see "Adjust the Amount of GC Threads for Processors" on page 2-6.

Basic Application Tuning

Use these guidelines when designing your applications for WebLogic Real Time.

- Understand your application code and how to measure latency.
- Avoid making synchronous calls to slow back-office systems as part of a transaction as this defeats the purpose of real-time. Conversely, make sure any non-critical calls are handled asynchronously through work thread pools, or by using JMS.
- Minimize memory allocation. If possible, allocate and free memory for a single transaction in a *chunk* as this helps avoid fragmentation of the Java heap. Also, minimize the amount and size of your objects.
- Control memory utilization by avoiding rampant memory allocation and allocating many large arrays.
- Free all objects as soon as possible; otherwise, objects that become unreferenced during a garbage collection might still be marked alive if they where referenced when the DetGC marked all live objects.

- Avoid long critical sections in your code, as synchronized blocks of Java code may cause a transaction to block.
- Avoid long linked structures; the deterministic GC needs to iterate through these objects.
- If transactions span more than one highly-active JVM, each such JVM may need to run Deterministic GC. For example, if a transaction is initiated by a Java client JVM, and the transaction includes both JMS server and J2EE server operations, all three JVMs may require Deterministic GC to reliably meet maximum latency criteria.

J2EE Application Tuning

Use these guidelines when tuning your J2EE applications for WebLogic Real Time.

- For server-side EJBs, MDBs, and Servlets ensure that there are enough concurrent instances configured to respond immediately to client requests (if all instances are active, this is a sign that client requests are queuing up behind each-other on the server).
- Make sure that resource pools contain enough instances so that threads are not forced to wait for resources. In J2EE for example, tune the EJB `max-beans-in-free-pool` property and tune thread pool sizes

JMS Application Tuning

Use these guidelines when using WebLogic JMS applications with WebLogic Real Time.

- Consider using asynchronous consumers rather than synchronous consumers.

  For more information on JMS consumers, see Best Practices for Application Design in *Programming WebLogic JMS*.

- Tune all JMS connection factory Messages Maximum settings to 1. This can potentially provide better latency at the expense of possibly lowering throughput. Similarly, configure your MDBs to refer to a custom connection factory with the following settings:
  - Messages Maximum = `1`
  - XA Connection Factory Enabled = `enabled`
  - Client Acknowledge Policy = `ACKNOWLEDGE_PREVIOUS`

  For more information on configuring JMS connection factories, see Configure connection factories in the *Administration Console Online Help*.

- For consumers of non-persistent messages from queues, consider using the WebLogic JMS `WLSession NO_ACKNOWLEDGE` extension.
- Ensure that your Spring JMS Templates leverage resource reference pooling (otherwise, they negatively impact response times as they implicitly create and close JMS connections, sessions, and producers once per message).

  Note: Resource reference pooling is not suitable if the target destination changes with each call, in which case change application code to use *regular* JMS and cache the JMS connections, sessions, producers, and consumers.

JVM Tuning for Real-Time Applications

These tuning suggestions can further improve performance and decrease pause times when using the JRockit deterministic garbage collector. For more information on the deterministic garbage collector, see the BEA JRockit Diagnostics Guide.

Allow For a Warm-up Period

There may be a *warm-up period* before response times achieve desired levels. During this warm-up, JRockit will optimize the critical code paths. The warm-up period is application and hardware dependent, as follows:

- For smaller applications (in terms of amount of Java code) with high loads that are running on fast hardware, there may be a warm-up period of one-to-three minutes.
- For large applications (in terms of amount of Java code) with low loads that are running on slow hardware (in particular, most SPARC hardware), there may be a warm-up period of approximately thirty minutes.

Adjust Min/Max Heap Sizes

Setting the minimum heap size (`-Xms`) smaller or the maximum heap size (`-Xmx`) larger affects how often garbage collection will occur and determines the approximate amount of live data an application can have. To begin with, try using the following heap sizes:

```
java -Xms1024m -Xmx1024m -XgcPrio:deterministic -XpauseTarget=30
```

For more information, see -X Command-line Options in the BEA JRockit *Reference Manual*.

Increase or Decrease Pause Targets

- If you specify `-Xgcprio:deterministic` without the `pauseTarget` option, it will be set to a default value, which in this release is 30 milliseconds.
- Running on slower hardware with a different heap size and/or with more live data may break the deterministic behavior. In these cases, you might need to increase the default pause time target (30 milliseconds) by using the `-XpauseTarget` option. The maximum allowable value for the `pauseTarget` option is currently 5000 milliseconds.
- Conversely, if you want to test your application for the lowest possible pause time, you can lower the default `-XpauseTarget` value down to a minimum value. In this release, the minimum value is 10 milliseconds.

For more information, see -X Command-line Options in the BEA JRockit *Reference Manual*.

Set the Page Size

Increasing the page size (`-XXlargePages`) can increase performance and lower pause times by limiting cache misses in the translation look-aside buffer (TLB). See -XX Command-line Options in the BEA JRockit *Reference Manual*.

Determine Optimal Load

Do not be overcautious in terms of load. The deterministic garbage collector can handle a fair amount of load without breaking its determinism guarantees. Too little load means the JVM's optimizer and GC heuristics have too little information to work with, resulting in sub-par performance. A best practice is to run your benchmarks at various loads (with and without deterministic GC) to determine the optimal load.

Analyze GC With JRockit Verbose Output

JRockit verbose output normally doesn't incur a measurable performance impact, and is quite useful for analyzing JVM memory and GC activity. Table 2-1 defines recommended verbose options for analyzing JVM memory and GC activity.

Table 2-1 JRockit Verbose Output Options

Option	What it does...
`-Xverbose:opt,memory,memdbg, gcpause,compact,license`	For GC and memory analysis.
`-Xverboselog:verbose-jrockit.log`	Redirects verbose output to the designated file.
`-Xverbosetimestamp`	Prints a formatted date before each verbose line.

Limit Amount of Finalizers and Reference Objects

Try to limit the amount of Finalizers and reference objects that are used, such as `Soft-`, `Weak-`, and `Phantom-` references. These types require special handling, and if they occur in large numbers then pause times can become longer than 30ms.

Adjust the GC Trigger

Try adjusting the garbage collection trigger (`-XXgctrigger`) to limit the amount of heap space used. This way, you can force the garbage collection to trigger more frequent garbage collections without modifying your applications. The garbage collection trigger is somewhat deterministic, since garbage collection starts each time the trigger limit is hit. See the BEA JRockit Diagnostics Guide.

Note: If the trigger value is set to low, the heap might get full before the garbage collection is finished, causing even longer pauses for threads since they have to wait for the garbage collection to complete before getting new memory. Typically, memory is always available since a portion of the heap is free and any pauses are just the small pauses when the garbage collection stops the Java application.

Adjust the Amount of GC Threads for Processors

With the variety of sophisticated processing hardware currently available (HyperTransport, Strands, Dual Core, etc.), JRockit may not be able to determine the appropriate number of GC threads it should start. The current recommendation is to start one thread per physical CPU; that is, one thread per chip not per core. However, having too many GC threads could affect the latency of applications since more threads will be running on the system, which might saturate

the CPUs, and thus affect the Java application. Conversely, setting them too low could increase the mark phase of the GC, since less parallelism is possible. For example, on a dual core Intel Woodcrest machine with four cores the recommended number of GC threads is two, which is the same as the number of processors in the machine.

To see how many GC threads that JRockit uses on your machine, start JRockit with `-verbose:memdbg` and then check for the following lines that are printed during startup:

```
[memdbg ] number of oc threads: <num>
[memdbg ] number of yc threads: <num>
```

If necessary, adjust the number of GC threads using the `-XXgcthreads:<# threads>` parameter.

For more information, see -XX Command-line Options in the BEA JRockit *Reference Manual*.

More Tuning Information

This section contains pointers to additional performance and tuning information.

JRockit JVM

- BEA JRockit Memory Management Basics contains information on all of the JRockit garbage collection options.
- About Profiling and Performance Tuning provides information on tuning the JRockit JVM.

See BEA JRockit Diagnostics Guide for additional diagnostic information about BEA JRockit.

WebLogic Server

- *WebLogic Server Performance and Tuning* contains information on monitoring and improving the performance of WebLogic Server applications.
- *Best Practices for Application Design* in *Programming WebLogic JMS* provides design options for WebLogic JMS application behaviors to consider during the design process, and recommended design patterns.