

# **Solutions for detect, diagnose and resolve performance problems in J2EE applications**

*Cristian Maties*

Quest Software

Custom-developed J2EE applications require performance assurance across the application lifecycle. Quest Software provides comprehensive coverage, from code optimization in development, through pre-production testing in QA, to 24x7 performance monitoring once the application goes into production.

We will present, how Quest Software provides an integrated solution designed to help all the stakeholders in J2EE application performance management accelerate the detection, diagnosis and resolution of business-threatening performance issues.

#### **Informacja o autorze:**

Cristian Maties – Senior Consultant having vast experience in a number of Quest Products.

## 1. Introduction

Businesses rely on Java 2, Enterprise Edition (J2EE) Application Servers to deliver highly reliable mission critical applications. These applications include self-service catalog services, real-time portfolio management, and 24 x 7 customer service. If these systems are not available, customers and money can be lost.

Monitoring solutions for J2EE applications need to aggregate data from multiple sources, provide a user-friendly interface to the data, and identify performance issues before customers experience poor response time or an outage.

This paper provides an overview of how J2EE addresses application management, identifies common hot spots to watch, and offers a solution to maximize the availability of your application by keeping you aware of potential problems.

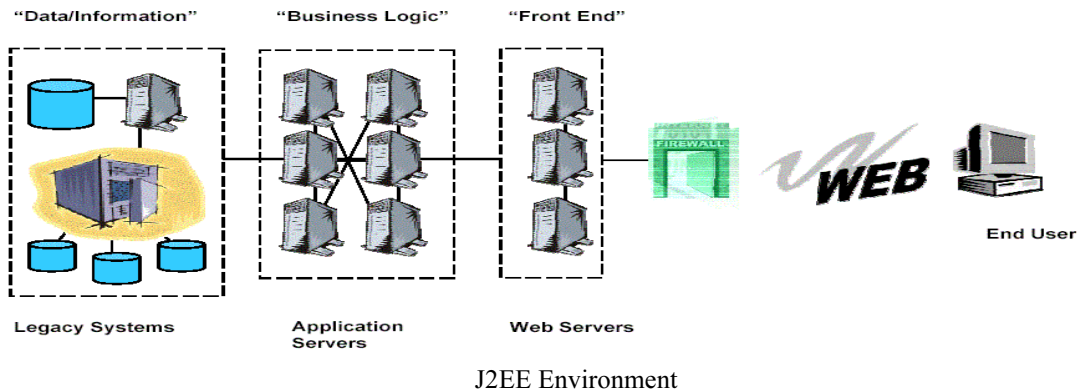
## 2. The Big Picture

J2EE was designed to provide an extensible platform for mission critical business applications. Developing n-tier J2EE applications provides much needed scalability, redundancy, and a separation between the customer interface and the business logic of an application. However these application environments also introduce additional complexity and new challenges for those charged with maximizing availability and performance.

Software development groups are at different stages of integrating J2EE technology. Customer Relationship Management (CRM) companies are extending their applications by adding HTML based interfaces. A J2EE application server is often introduced at this time to host Server side Java (servlets) and Java Server Pages (JSPs).

When the product architect determines that the application requires a greater degree of management, he can design the application to take advantage of the persistence and transactional framework provided by the J2EE application server. The server manages the resources used by J2EE applications including memory, database connections, thread pools, and caching.

Once deployed the J2EE application consists of several tiers. Clients use Web browsers to access a Web server. The Web server may process the request independently or it may pass the request back to an application server. If the necessary data is cached locally, the application server sends a response back through the Web server to the client. Otherwise, the application server queries remote databases and legacy systems in order to aggregate a response to the customer query.



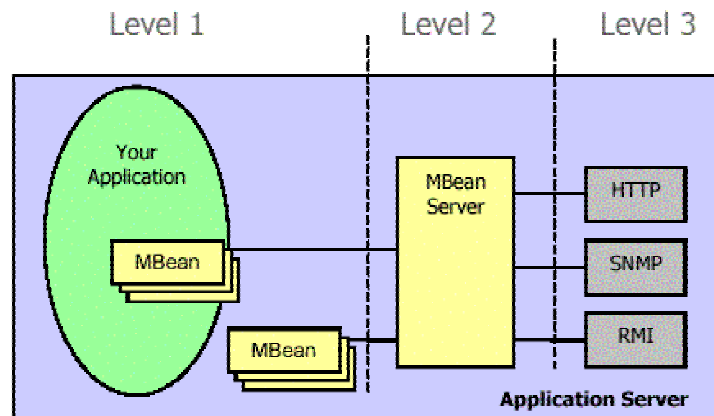
### 3. Data Collection

Fundamental to every monitoring solution is the data. Without quality information available to the system, warnings and alerts cannot be generated. Performance monitors for J2EE applications need to aggregate data from multiple sources. The first data source is the application server itself. In addition to providing a platform for applications, the server also provides a management framework. Recent application servers from all the major vendors, including BEA, IBM, Oracle, and SUN, implement the standard framework called Java Management Extensions (JMX).

The JMX framework consists of three levels. At the first level are Managed Beans (MBeans). MBeans expose configuration data and methods to change configuration values. MBeans also provide current resource usage. For example, an MBean can tell you the maximum size of an EJB cache, the current size of the cache, and the ability to change the maximum size.

MBeans are managed by the MBean Server that reside at the second level of the JMX framework. The MBean server contains the registry for MBeans and provides services to manipulate them. In the above example, remote applications go through the MBean Server to inspect and manage the EJB cache size.

The final layer consists of JMX adapters that assist external applications to access the MBeans. This layer is specified as part of the JMX standard but its implementation is not required. As such application server and software vendors write adapters to meet their specific needs. For example, Web-based console used to monitor the EJB cache size would use the HTTP adapter to access the MBean Server.



*Three Levels of the JMX Framework*

Through the JMX framework, application server vendors provide access to the current resource usage and configuration of the EJB and Servlet containers. While this is a standard framework, application server vendors are free to choose which attributes of their application servers are exposed. Resources commonly monitored through JMX include EJB usage, transactions, thread pools, servlet pools, JMS thresholds, and cache sizes.

In addition to variations between application servers, there is a significant amount of information pertaining to the application that is not typically available through JMX. The process, known as instrumentation, is necessary to obtain information not provided by the vendors including details about the individual methods that make up the application. There are two types of instrumentation used to obtain method level data. The first is designed by the application architect and implemented by the programmer during the development of the application. Once compiled, the application exposes performance information as specified by the architect.

Software monitoring tools do not usually have access to application source code during the design stage. The monitoring solution is typically selected as the application enters the load testing stage prior to deployment. Adding instrumentation to the source code at this time would introduce risk to the project and considerable delay to the release while the changes are made and tested. Instead of modifying the source code, monitoring tools typically apply a second form of instrumentation to the byte-code of the compiled application. By adding a small amount of additional byte-code around compiled methods, the necessary performance information is exposed to the monitoring system.

While JMX can be used to determine many attributes of the EJBs, it cannot expose everything that happens inside of the EJB. Byte-code instrumentation occurs within the bean allowing for a deeper level of runtime data than when JMX is used alone. Once instrumented, class and method data is available including response time distributions, and usage counts, and thrown exceptions.

Individually, JMX and byte-code instrumentation provide valuable insight into the J2EE application server and the hosted applications. Using data from both sources, monitoring tools are able to report an accurate picture of the availability and performance of J2EE applications and alert administrators of potential problems.

## 4. Common Hot Spots

### Garbage Collection

The manner in which your application uses memory can greatly affect its performance; something as simple as creating a new instance of an event object and passing it between the Web-tier and EJB-tier can appear harmless during development and unit testing, but under load testing the memory impact can be significant. For example, if each request uses 10K of memory, multiply that by 500 simultaneous users making requests on the average of 5 seconds each: after running for 5 minutes, the memory allocated for this 10K object is 300MB. Combine this with all of the other objects you are creating and suddenly garbage collection becomes a major issue.

One of the benefits of Java is that the virtual machine manages all of your memory for you: this is both a blessing as well as a curse because while you are not burdened with the task of memory management, you cannot explicitly manage it either. Thus the Java Virtual Machine (JVM) maintains a thread that watches memory and reclaims memory as needed. There are

various virtual machines, but for the purposes of this discussion we will focus on the Sun Java Virtual Machine version 1.3.1 (as it is still shipping with most production application servers).

The Sun JVM manages memory by maintaining two separate generational spaces that objects can be allocated in: the young generation and the old generation. By managing a young generation, the JVM can take advantage of the fact that most objects are very short lived and are eligible for garbage collection very shortly after they are created; the young generation runs very quickly and efficiently as it either reclaims unused memory or moves old objects to the old generation using a copying mechanism. There are three types of garbage collection that it supports:

- Copying (or scavenge): efficiently moves objects between generations; default for minor collections
- Mark-compact: collects memory in place but significantly slower than copying; default for major collections
- Incremental (or train): collects memory on an on going basis to minimize the amount of time spent in a single collection; you must explicitly enable using the “-Xincgc” command line argument

The default behavior of garbage collection is to reclaim the memory that it can by copying objects between generations (minor collections) and when the memory usage approaches the maximum configured size then it performs a mark-compact operation (major collection). In a single application environment a major collection slows down the application, but runs very rarely; in an enterprise application, however, we saw that operations performed on a simple request generated 300MB of memory usage in a 5-minute period. A major collection is catastrophic to the performance of your application server. Some major collections can take in upwards of a few minutes to run and during that time your server is unresponsive and may flat out reject incoming connection requests.

So how can you avoid major collections? Or if you cannot, how can you minimize their impact on your system?

Tuning the JVM involves a couple steps:

1. Choose a heap size that supports your application running your transactions under your user load
2. Size your generations to maximize minor collections

The default behavior of the JVM works great for stand-alone applications, but abysmally for enterprise applications; the default sizes vary from operating system to operating system, but regardless the performance is not tuned for the enterprise. Consider the JVM running under Solaris, it has a maximum heap size of 64MB of which 32MB is allocated to the young generation. Consider allocating 300MB in 5 minutes, or 60MB per minute with a total heap size of 64MB - and recall that your requests are not the only thing running in the virtual machine. The bottom line is that this heap is far too small. A rule of thumb is to give the virtual machine all of the memory that you can afford to give it.

When sizing the generations we must take a closer look at the structure of the young generation: it has a region where objects are created called Eden and defines two survivor spaces; one survivor is always empty and is the target for the subsequent copy - objects are copied between the survivor spaces until they age enough to be tenured to the old generation. Properly sizing the survivor spaces is very important because if they are too small then the minor collection cannot copy all of the required objects from Eden to the survivors which causing it to run much more frequent and forces it to prematurely tenure objects. Under Solaris the default size of the

survivor spaces is 1/27th of the entire size of the young generation, which is probably too small for most enterprise applications.

The next question is how to size the young generation itself. Unless you are experiencing problems with frequent major collections, the best practice is to allocate as much memory to the young generation as possible up to half the size of the heap.

After making these changes, watch the heap and the behavior of garbage collection while the system is under load and adjust the sizes accordingly.

## Entity Bean Cache

Entity Beans define an object model running between your application and your persistent data; Entity Beans usually communicate with databases although they are not limited to only database communication. For the purposes of this discussion let us assume that our Entity Beans are communicating to a database, but the discussion is equally applicable to legacy systems or other implementations.

In a properly built enterprise Java application, you will delegate your data persistence to Entity Beans, not only because that was their intended purpose from the inception of J2EE, but because application servers provide a caching mechanism that manages your Entity Beans for you. The lifecycle of a data request is as follows:

1. A component requests and Entity Bean
2. The Entity Bean is loaded and initialized with the persistent information it represents
3. The Entity Bean is stored in the Entity Bean cache for future reference (activated to the cache)
4. The Entity Bean's remote (or local) interface is returned to the requesting component

Subsequent requests will be serviced directly from the Entity Bean Cache and will bypass the creation of the object and its initialization (a query to the database); thus database access is minimized yielding significantly enhanced performance. The nature of a cache is that it has a predefined size specifying how many objects it can hold and then it manages the life times of those objects based off of an internal algorithm. For example, it might keep the most recently accessed objects in the cache and remove objects that are seldom accessed to make room for new objects. Since caches have predefined sizes, the sizing of the cache has a significant impact on performance.

When referring to Entity Bean Caches, the term for loading an object from persistent storage into the cache is called activation and the term for persisting an object from the cache to persistent storage is called passivation. Entity Beans are activated into the cache until the cache is full and then, when new Entity Beans are requested, some existing beans must be passivated so that the new beans can be activated. Activating and passivating beans creates an overhead on the system and, if performed excessively, actually eliminates all of the benefits of having a cache. This state of excessive activations and passivations is referred to as thrashing.

When tuning the size of your Entity Bean Cache, the goal is to size it to service most of the requests from the cache, thus minimizing thrashing. As with every tunable parameter there is a trade off: a large cache requires more system resources (memory) than a small cache. So the other facet of your goal is to ensure that the cache is as large as it needs to be but not much larger.

The tuning process is to load test your application using representative transactions and observe, using a monitoring tool, the behavior of your Entity Bean Caches, paying particular attention to:

- Number of Requests
- Number of Requests serviced by the cache (hit count)
- Number of Passivations
- Number of Activations

If you see excessive activations and passivations then increase your cache size until they are few or non-existent. The result will be shorter request response times and diminished database accesses.

## Segmentation

Segmentation of resources is the act of assigning specific application components to use specific sets of resources. Segmentation can be applied to both thread pools as well as JDBC connection pools.

All application servers maintain thread pools that are used to service incoming requests; in WebLogic these thread pools are called execute threads and contained within one or more execute queues while WebSphere terms them thread pools. Regardless of the implementation, a request that arrives at the application server is queued to wait for or directly dispatched to a thread for processing. When the thread is available, it takes the request and performs some business logic and returns a response to the caller. In most application servers, the default behavior is to assign all applications and components to be serviced by the same pool of threads. If all components in your application are of equal business value and one business process is permitted to wait for an indefinite amount of time because another business process is in use, then this is an acceptable model. In practical application however, certain business processes have more intrinsic value than others and regardless one business process should not inhibit the performance of another business process. Consider deploying the following set of components:

- E-Commerce Store Front-End
- E-Commerce Checkout and Billing Component
- Administration Component

Each of these three components has a specific purpose and value to the business: the front-end allows customers to browse the company's products, compare prices, and add components to his shopping cart; the checkout and billing component is responsible for gathering customer demographic and credit card information, connecting to a credit card billing server to attain purchase confirmation and debiting the customer's credit card, and store the record in the database for reference; the administration component is your gateway to manage your e-store, track orders, and manage inventory. Since each has a specific purpose and value to the business, each must have the opportunity to execute in an efficient manner. For example, a customer browsing the store should not inhibit another customer from placing an order and likewise the customer placing an order should not slow down customers browsing the store; browsers must become buyers and buyers must complete their transactions for the company to be able to make any money. Finally the administration component must be able to run concurrently both of the other components in case problems develop; for example if the credit card billing company changes its online address or if the company's account is renewed, the change update must be seamless to the customer.

How can you ensure that each component has its fair access to the system resources so that it can complete its task?

The answer is to create individual thread pools for each component that only it has access to at run-time. The size of each thread pool needs to be tuned to meet the requirements of the system and the user load; more frequently access components, such as the front-end, will need more threads than the checkout and billing, which will probably need more threads than the administration component. If the system is under severe load and the front-end is running at capacity, orders can still be placed and the store administrator can always connect to the administration component.

This concept can be further extrapolated and addressed at the database connection pool level. Even if all of your underlying persistence is stored in one database, consider creating multiple connection pools to that database, each categorized by its business function. Components that are browsing the store should be serviced by one connection pool while another services customers placing orders. This will avoid contention for database resources between two components trying to share a single connection pool; again you do not want your store browsers to inhibit other customers from paying you. Segmentation of threads and database connection pools into logical business functions can help ensure your application's response under load and guarantee that critical processes are properly serviced. The result is that the tuning overhead is greater, because you have more things to tune, but the end user experience, which is the true goal of performance tuning, will be greatly enhanced.

## 5. Quest's Solutions

The Application Administrator is charged with the care and feeding of the J2EE application servers and it is the administrator who performs the initial triage when a problem is reported. Problems typically originate when the end user initiates a trouble ticket because she is unable to complete a transaction. With little more than the trouble ticket, the administrator needs to isolate the problem and expedite a recovery process.

It is imperative to quickly uncover where performance bottlenecks exist at this stage of the triage process. To find the problem, the administrator begins with a birds-eye view of the application environment. This environment consists of many nodes including Web servers, application servers, databases, and a variety of legacy systems. Foglight®,

Quest Software's application performance monitor, enables the Application Administrator instant access to the status of each of these nodes.

The screenshot shows the Foglight IP Map interface. The window title is "User foglight connected to aspen.pls.quest.com (IPmap)". The interface includes a menu bar (File, Edit, Add, View, Tools, Wizards, Help) and a toolbar with various icons. The main area displays a tree view of IP addresses and their associated events and statuses. The tree is organized into several IP ranges: 10.X.X.X, 127.X.X.X, and 192.X.X.X. Each IP address is listed with its status and a description. The status column contains colored boxes with numbers indicating the count of events: red for Fatal, yellow for Foglight Daemon Down, and blue for Critical. The description column provides details about the events, such as "Fatal Event(s)", "Foglight Daemon Down", and "Critical Event(s)", along with the operating system version.

Name	Events	Status	Description
IPmap		Fatal Event(s)	
10.X.X.X		Fatal Event(s)	
10.4.X.X		Fatal Event(s)	
10.4.38.X		Fatal Event(s)	
aspen	2	2	Fatal Event(s) SunOS5.8
bookstore2		2	Foglight Daemon Down SunOS5.8
jupiter1			Foglight Daemon Down Windows2000
127.X.X.X		Nominal	
127.0.X.X		Nominal	
127.0.0.X		Nominal	
bookstore1			Foglight Daemon Down Linux2.4
192.X.X.X		Fatal Event(s)	
192.168.X.X		Fatal Event(s)	
192.168.40.X		Fatal Event(s)	
despair	1	1	Foglight Daemon Down SunOS5.7
desperado			Ok SunOS5.8
manila	1	2	Fatal Event(s) SunOS5.8
passion		1	Critical Event(s) SunOS5.7

*Foglight IP Map*

Foglight's Cartridges for J2EE application servers convert the numerous data points collected through JMX and instrumentation into actionable information and presents the results in a user-friendly interface.

To expedite problem identification, Foglight for J2EE enables the administrator to drill down through domains and clusters to individual servers in order to identify if there is a problem with a server or group of servers. Foglight cartridges come with domain specific charts and data views to further isolate performance bottlenecks.

## Intuitive Alerts

Foglight cuts through the clutter to provide 24 x 7 unattended monitoring. Customized rules for J2EE applications filter through the data collected through JMX, instrumentation, and log files to alert the Application Administrator when a problem is detected. The rule editor is based on performance variables set to meet customer requirements and can be reused across multiple rules. Rules trigger alerts across a range of conditions. By default, alerts appear on the Foglight console. Alerts may also be sent through email to one or more recipients based on the severity of the alert. Each alert message contains the severity and description of the problem. From the alert, the administrator can drill down to the alert detail and access the help system for further information.

Standard rules in Foglight monitor the availability and performance of J2EE applications and application server resources. Rules can watch for a threshold to be crossed such as when the number of active threads reaches 100 or the number of available application servers drops from ten down to four. Instead of fixed values, a rule could trigger an alert when 90% of the JDBC connections are in use or when the number of available servers drops to 40%. A third type of rule can compare the current resource usage to the average resource usage and trigger an alert when necessary. For example, Foglight can send an email alert when the average re-

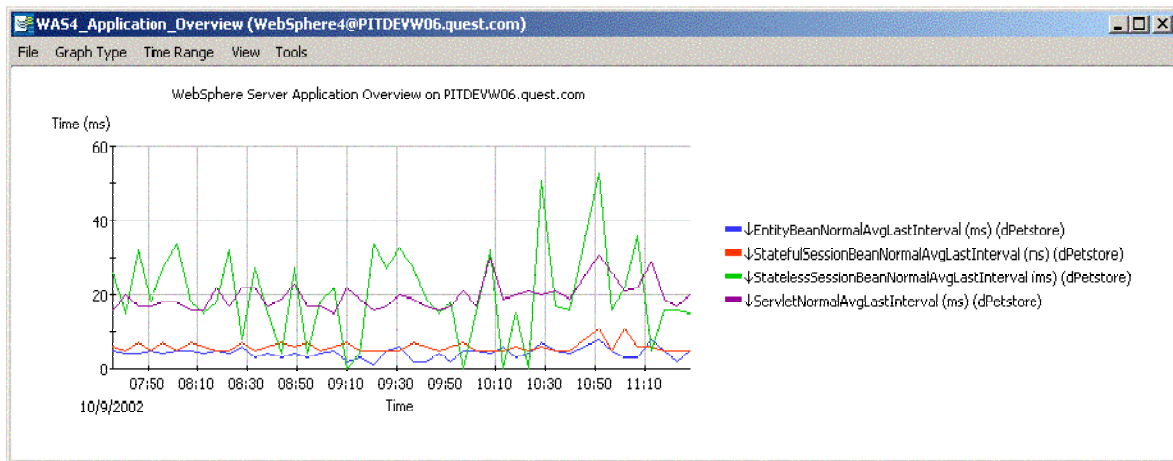
sponse time of a servlet is 50% longer than the running average. This type of rule learns the typical behavior of the application and alerts the administrator when the application responds abnormally.

## Chart and Data Views

Foglight views of the data collected through JMX and instrumentation are available for J2EE applications. Easy to read charts provide quick access to the resource usage, availability, and performance of EJBs and Servlets. Additional charts can be created, stored, and shared. Any report can be formatted as a report and sent via email.

Below are 20 common questions answered by Foglight charts and data views.

1. What has my server availability been for the past week?
2. How many servers in my cluster (workgroup) are available?
3. How many Entity Beans are active (cached) in each application?
4. How many Session Beans are active in each application?
5. How many threads are in use in each execute queue?
6. How much memory is being reclaimed during Garbage Collection?
7. How many JDBC connections are currently in use?
8. What is the overall usage of the JMS server?
9. What is the message and byte usage for all JMS Topics and Queues?
10. How many HTTP Sessions are currently active for each application?
11. What has the JVM heap availability been for the past 24 hours?
12. How many transactions have run through the system?
13. Have there been a large number of application rollbacks today?
14. How many rollbacks were caused by a JTA Resource?
15. What is the average response time for each monitored Class?
16. What is the average response time for each monitored Method?
17. Which method is causing my class to respond abnormally?
18. For a given method, how do the latest response times compare to the average?
19. What is the average response time for each monitored Servlet?
20. What is the number of invocations for each monitored Servlet?



*Average Response Times for EJBs (drill-down available)*

## 6. Summary

The number of J2EE applications continues to increase as vendors migrate to the J2EE platform from their homegrown applications and as IT departments standardize on Java as their preferred development environment. Programmers and architects, who filled in as application administrators for several years, do not have the time to sit with production systems. Custom scripts and point products are no longer adequate to ensure the high availability required by today's customers. A comprehensive solution that analyzes critical data across all tiers of the application environment is must for every company. Foglight and the Foglight Cartridges for J2EE application servers provide the 24 x 7 unattended monitoring required by companies that wish to provide exceptional service to their customers.