# Using Patrol for BEA WebLogic to Accelerate the Value of WebLogic Server (WLS) Tuning

The purpose of this paper is to explain how a customer of Patrol for BEA WebLogic from BMC Software can use the product to monitor the performance of BEA WebLogic Server (WLS). There are many BEA WebLogic performance-related configuration parameters available to the administrator who is interested in streamlining the efficiency of the Server. This paper attempts to demonstrate that focusing on a subset of those parameters will enable effective and rapid tuning and produce improved application throughput. The tuning information presented in this paper is based upon:

a) WebLogic Server performance recommendations and instructions from BEA Systems as described in the WebLogic Server Performance and Tuning Guide;
b) field observations with customers;
c) experimentation in a lab environment;
d) Course material from BEA Education Services – Administration of BEA WebLogic Server;
e) Java documentation from Sun Microsystems. This paper is meant to be complementary documentation to the Patrol for BEA WebLogic User Guide.

### Primary Tuning Considerations

Performance of a WebLogic Server system is rated by its response time, latency, and throughput. Four main considerations, if carefully handled, will deliver information that can be used by administrators to provide stability and improve application throughput. These are:

- Execute Queues
- JVM Heap size
- JDBC Connection Pool efficiencies
- JMS Server throughput

## Execute Queues

Consider those times when you and four hundred other travelers are queued up for Customs inspection at a major international airport. You observe there are only twelve inspection booths open out of twenty-two. The question that simmers in the minds of people is *why don't they simply open the other booths*? Like any other workflow process the airport customs procedure is not only dependent upon properly configured resource settings at that point (i.e. the number of inspectors), it is also linked and dependent

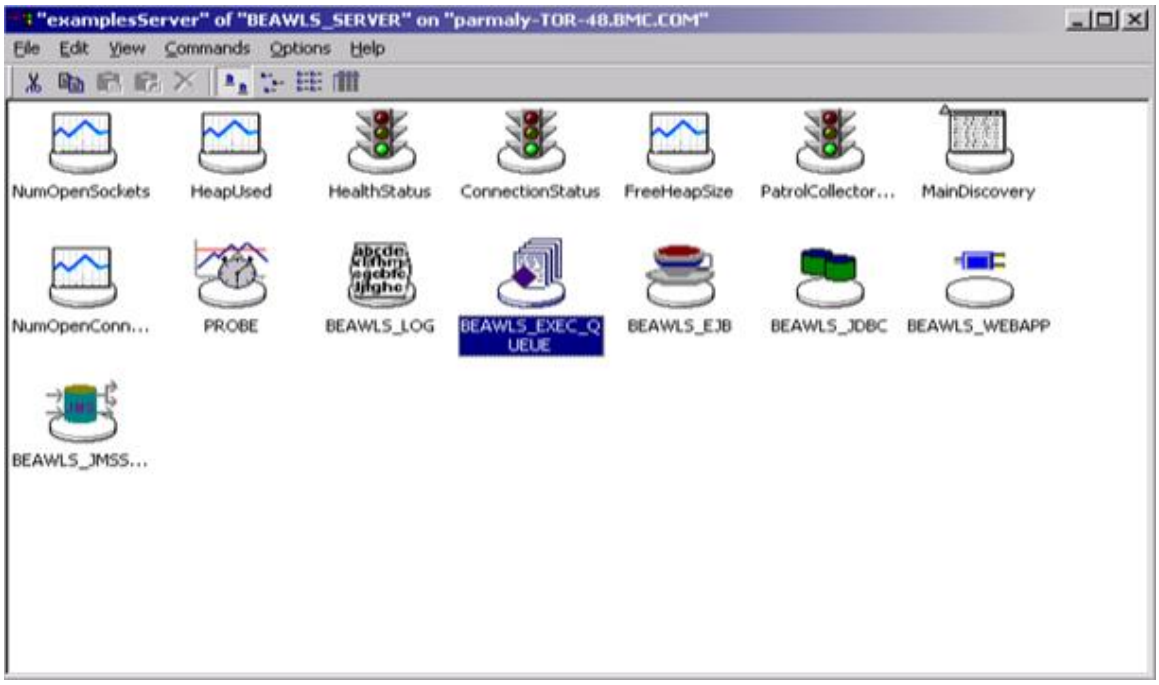
upon other properly configured resource settings for processes downstream. Too many inspection booths open can cause a backlog at some secondary step (i.e. mechanical processes such as escalators, secondary customs inspections, baggage handling, etc.). At London's Heathrow Airport, for example, the crush of passengers necessitates a tight control over the flow via staffed checkpoints at a variety of other locations aside from just the customs inspection point. The various checkpoints communicate with each other to ensure an orderly flow of passengers. If one checkpoint were to process passengers at too quick a rate, it would cause a backlog somewhere else in the facility. To increase the flow of travelers through the customs checkpoints will certainly provide immediate relief for passengers but will not necessarily decrease the total time it takes one to reach the departure gate.

**WebLogic things to remember - #1:**
Use the Execute Queues not just as a means to segregate applications. Use them as levers to control the flow of requests that the WebLogic Server must handle within the context of other work that the Server must perform.

The role of the WebLogic Execute Queues is to act as a conduit for work that needs to be executed by the WebLogic Application Server. Too much throughput through too many Execute Queues could have a deleterious effect on Server performance. Because WebLogic Server performance is only as good as how well the JVM performs and if the JVM is not properly sized, WebLogic Server will not be able to handle the incoming work requests. Each work request uses a thread that is invoked from an execution queue. Threads consume memory. Too many threads can mean too much of the JVM's available memory is used consequently forcing the Server to perform needless I/O.

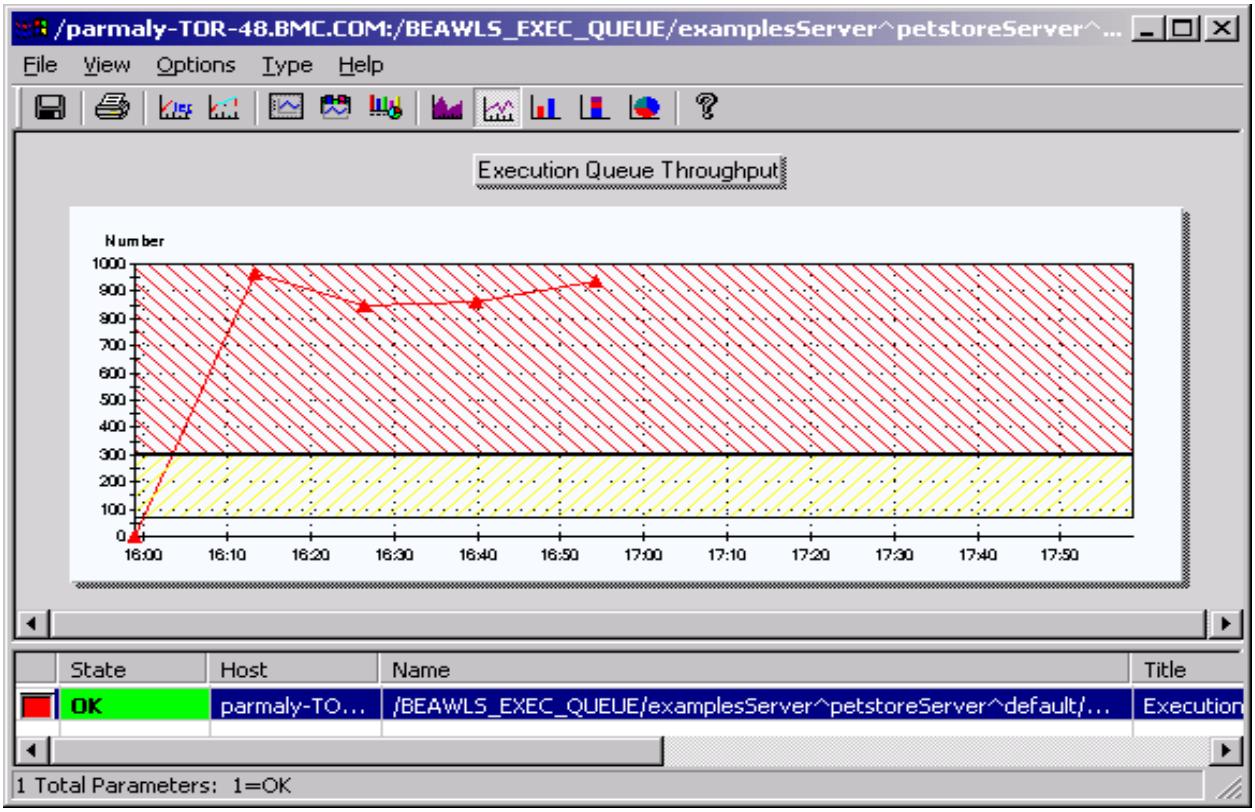
Administrators need to treat Execute Queues as throttles or levers for controlling the performance of the WebLogic Application Server. Additionally, the work that needs to be executed can and probably should be categorized according to applications or the criticality of the tasks, thus providing an even stronger method for controlling the flow of work to be performed but down to the application level. In the airport analogy, this level of service would be delivered through the creation of separate queues to expedite passage for frequent pre-authorized travelers or flight crews. Expediting the processing of "trusted" passengers optimizes the available resources (inspectors) by making them more readily available to deal with those passengers who require more time for processing. Similarly, through careful settings within WebLogic the Execute Queues can be managed to optimize the throughput of trusted applications.

Use the Patrol's BEAWLS_EXEC_QUEUE application class to report on the various Execute Queues.




Execute Queues interoperate with the WebLogic Server through the use of threads. Threads carry the requests that need to be executed by the application server. Close examination of the Execute Queues will provide an administrator with the information necessary to properly configure the maximum number of threads available for specific applications. Therefore, the administrator can control the throttle of the Execute Queues by controlling the number of threads assigned to each queue.

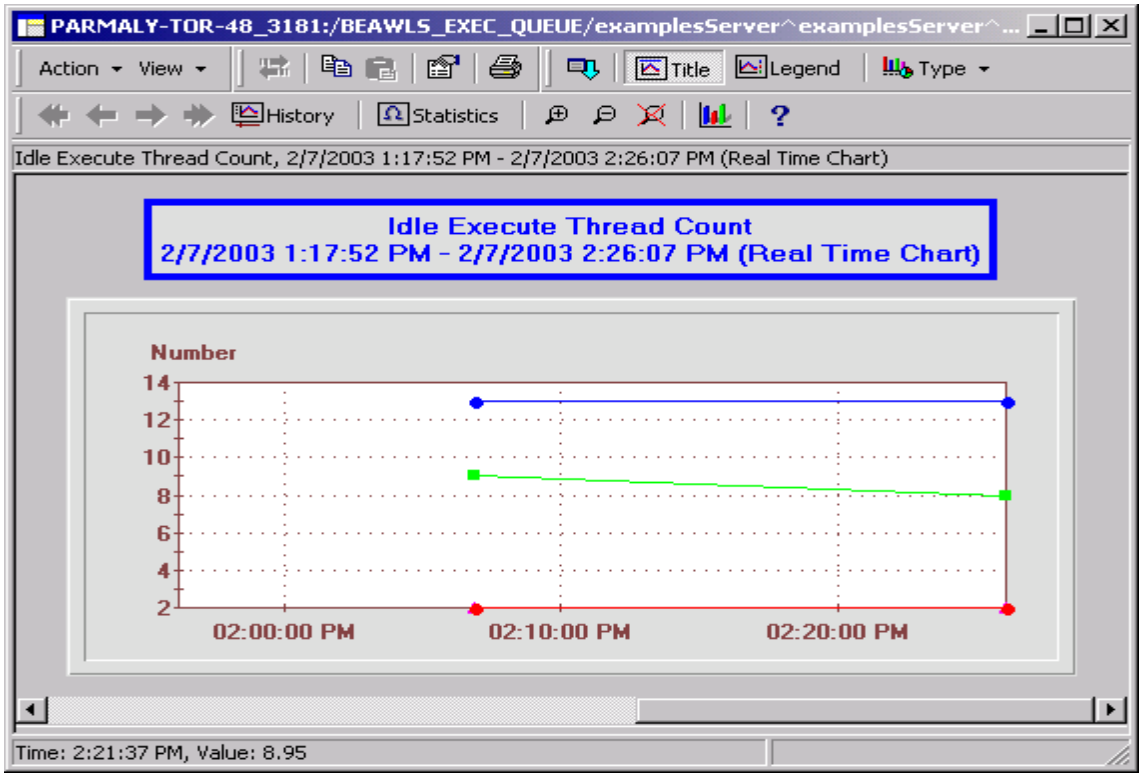
My WebLogic implementation (lab) has four Application Server instances defined, each with its own set of Execute Queues. In the sample graph on the next page, I generated activity using the *PetStore* application that is shipped with *BEA WebLogic,* and that runs on the *PetStore* application server that is also shipped with the product. I signed in to the online store, loaded up my shopping cart, checked out, signed out, registered and signed in as a new customer, shopped for more pets, reviewed what I had selected and updated the amount, and finally checked out again. I inspected the Patrol console to see what was reported for that last five minutes of activity by the Execution Queue Throughput parameter. Execute Queues do not discriminate between objects so they handle all types of requests being submitted to the application server. Therefore, the queues will contain numbers that relate to all objects that need to be processed by the Application Server (EJBs, Servlets, JSPs, etc.). The graph can therefore display very high numbers, more than the user might think **for what the application seems to deliver.**




One can see that the shopping activity of a single user drove the throughput rate from zero to 1000 in a very brief period of time. That type of activity burst is very common in the Internet commerce world and administrators need to understand not only the nature of the application but also how WebLogic Server is designed. An understanding of the architecture will inform the administrator of how Execute Queues interoperate and are interdependent with other WebLogic Server components. As stated earlier, Execute Queues consume memory through the use of threads. Allocating a large number of threads for applications with the goal of improving application performance could have an unintended adverse affect. Threads that are simply waiting for work (known as Idle Threads) will rob the server of valuable memory thus degrading overall server performance. This is another reason why it is important to report on the Execute Queues. The administrator must not only consider facilitating individual application throughput, they must also be mindful of how that effort affects the performance of WebLogic Server as a whole.

From a monitoring perspective, it would be helpful to be able to compare the activity levels of various Execute Queues that operate with a specific WebLogic Server instance. In the example below, I used the Patrol console and dragged and dropped three separate parameters for **Idle Execute Thread Count** from three Execute Queues and dropped them into the same graph. The resulting combined graph provides the administrator with a comparative view of how well the three queues are being used in relation to each other. A high Idle Thread Count relative to the others (meaning that the queue is not busy) should indicate that the number of threads for that queue could be scaled back or perhaps the queue should be eliminated entirely and all that application's

work be sent through the default Execution Queue. To be certain the administrator should only base this determination on a day's worth of data. Applications will typically report varying levels of activity throughout a 24-hour period. This simple graph can result in a reclaiming of memory thus making it available to other WebLogic components.




**WebLogic things to remember - #2:**
Look for **high** volume rates and a **low** Idle Thread Count through the Execution Queue for they signal a well-optimized queue.

## JVM HEAP Size

WebLogic Server runs in an instance of the Java Virtual Machine. The JVM executes bytecodes in Java class files. JVM Heap size is the amount of memory used by the WebLogic Server on that CPU. Memory is the fundamental method that transaction managers, database managers, and application servers employ to deliver speedy response time. By caching frequently used objects in memory, WebLogic Server delivers quick request turnaround because it satisfies application data and object requirements by finding what it needs in memory without having to incur the overhead associated with I/O to disk. It is a double-edged sword though. Too much memory allocated to the

JVM will seriously degrade the application server's performance because it will make memory unavailable to other memory-dependent components (Execute Queues, connection pools). Conversely, if not enough memory is allocated to the JVM, WebLogic performance will suffer as too much time will be spent executing I/O to refresh data rather than being able to find the necessary objects in memory or cache. I/O means slower response time. This is a critical performance consideration and striking an ongoing balance is the best method for improving the overall performance of WebLogic Server.

**WebLogic things to remember - #3:**
A high value for the HeapSize parameter should be the first clue for the administrator to look for a) too much garbage in the JVM or; b) high activity rates for the Execute Queues and/or; c) high activity through the JMS Server. Adjusting garbage collection intervals, or the number of threads assigned to the Execute Queues, or the number of messages handled by the JMS Server may result in lower Heap Size use by the JVM without having to resort to increasing the amount of memory allocated to the JVM.

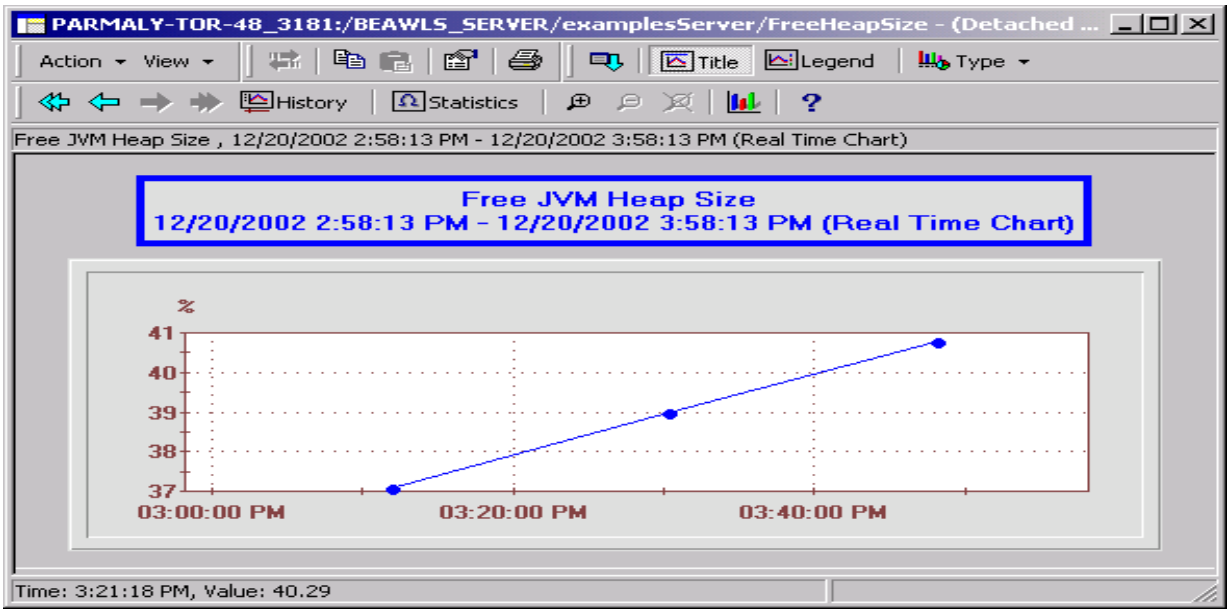
**What causes garbage to build up in the WebLogic JVM?**
In order to deliver optimal throughput and response time WebLogic uses memory to store frequently used objects. These objects will stay in memory even after the applications that called them are no longer active. An object is therefore considered garbage when there are no pointers to it from a running program. If memory is left unattended it will quickly fill and WebLogic Server will not be able to deliver the expected service. Therefore, periodic garbage collection is something that WebLogic executes to clear the JVM of objects that are no longer required. The administrator has the capability to tune the interval time between collections.

**WebLogic things to remember - #4:**
The goal of tuning heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time.

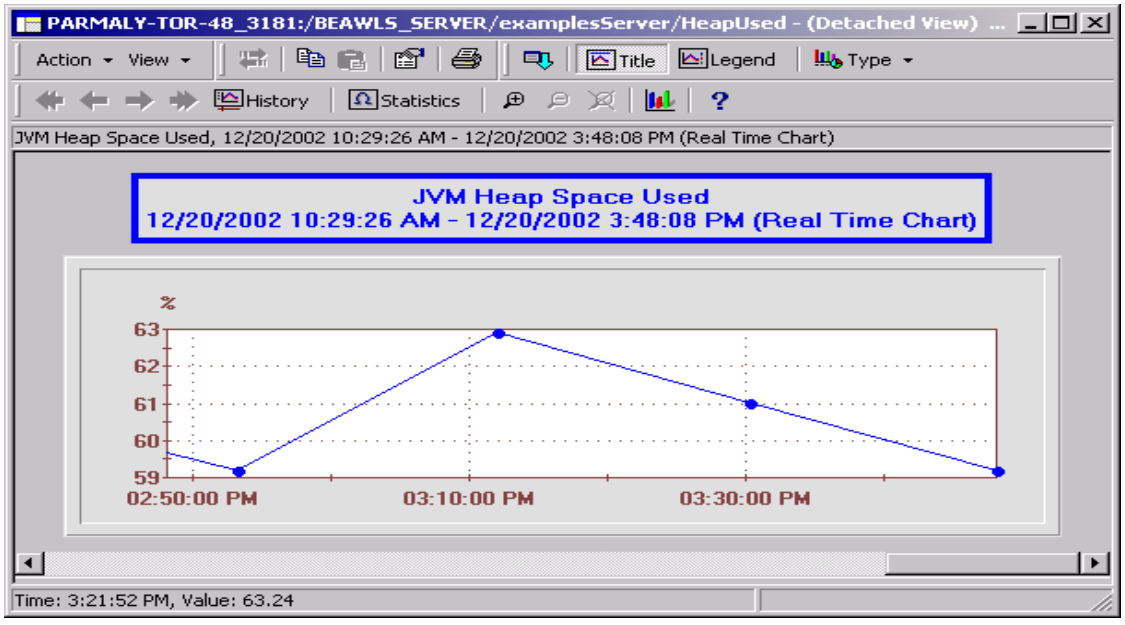
Garbage collection is resource intensive for the WebLogic server, greatly degrading performance when it occurs. An optimally tuned Heap size will either minimize the frequency of garbage collection or shorten the time it takes to do the collection thus freeing the WebLogic server to be available to service more clients. Larger heap sizes mean less frequent collections but they will run longer; smaller heap sizes mean more frequent collections that run faster. Both strategies have their advantages.

The following two graphs provide information regarding the health of the JVM Heap space. The first, **Free JVM Heap Size**, enables a WebLogic administrator to monitor whether the JVM is optimally configured at certain times of the day. If the graph shows

the free space is trending towards a very low number (5% is the default low value) it indicates that WebLogic is at risk of running out of memory and will need to resort to performing I/O to satisfy requests. As stated, I/O is very expensive in terms of time and resources.

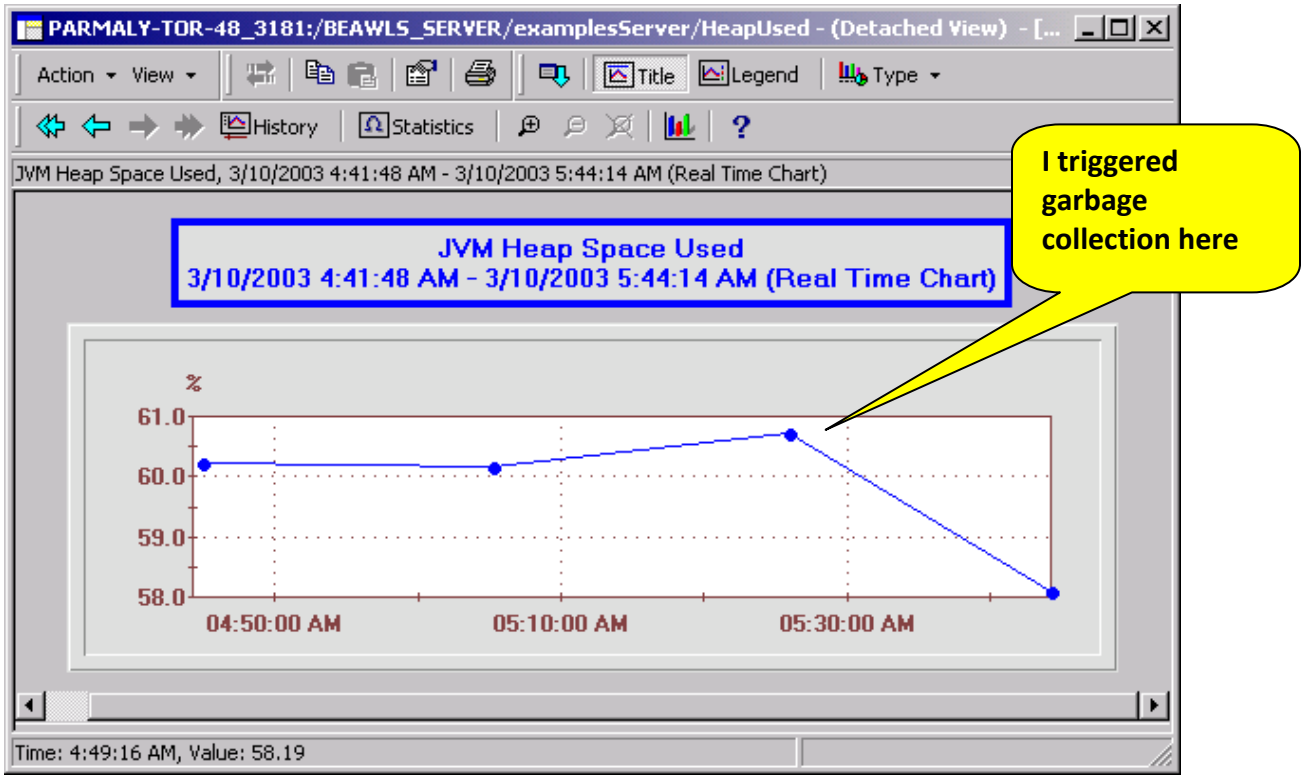



The next graph shows the total amount of heap used by WebLogic and all its applications. Careful monitoring of this information will enable an administrator to deliver a highly tuned WebLogic environment. Thresholds should be set to warn and alert when the Heap space used exceeds a certain amount for a certain period of time. Again, the goal of JVM Heap space monitoring is to strike a balance between the amount of memory allocated to WebLogic and the amount it actually uses.




Although Patrol for BEA WebLogic has a KM command to perform garbage collection, this action should be avoided unless full garbage collection is required. See the graph

immediately below and the amount of memory reclaimed after the garbage collection action was invoked from the Patrol console.




Setting up automated garbage collection based on accurate thresholds is a critical consideration since the alternative (manual triggering of garbage collection) can have severe consequences for WebLogic Server performance.

BEA WebLogic Server exploits the Java 1.3 JVM HotSpot capability called Generational Garbage Collection. With that feature, garbage collection is performed based on the lifetime of an object. Whereas when manual collection is triggered the JVM will examine ALL objects in the Heap to determine which ones are dead and should be removed. This has a seriously negative effect on WebLogic Server performance because it will halt processing while collection occurs. Generational Garbage Collection on the other hand divides objects into Young and Old. The Young category is further divided into Eden and two Survivor spaces. New objects are created in the Eden space and when garbage collection occurs, those objects are copied into one of the two survivor spaces. When the next collection occurs, those objects are copied into the next survivor space. This process continues until the JVM Heap Size exceeds a predefined threshold at which point the objects are moved into the Old category. The advantage of Generational Garbage Collection is that the actual process of garbage collection is quicker because unlike regular garbage collection, it does not examine each object in the JVM. When Generational Garbage Collection is employed, WebLogic Server performs much of that analysis in advance of the collection and only the Old category is

cleared by the garbage collection. This is a much faster and less-resource intensive process than the pre-HotSpot method.

With that information in mind it is critical that the garbage collection rate be monitored to ensure that the frequency is sufficient to replenish memory before the JVM resorts to executing I/O to satisfy requests. The Patrol for BEA WebLogic user can monitor garbage collection rates through the **Garbage Collection Count** and the **Garbage Collection Time** parameters within the BEAWLS_JVMPROFILER application class.

**Note from Peter: Here I would like to add graphs showing the parameters mentioned above as well as corresponding graphs showing the impact that those garbage collections had on the JVM Heap Size.**

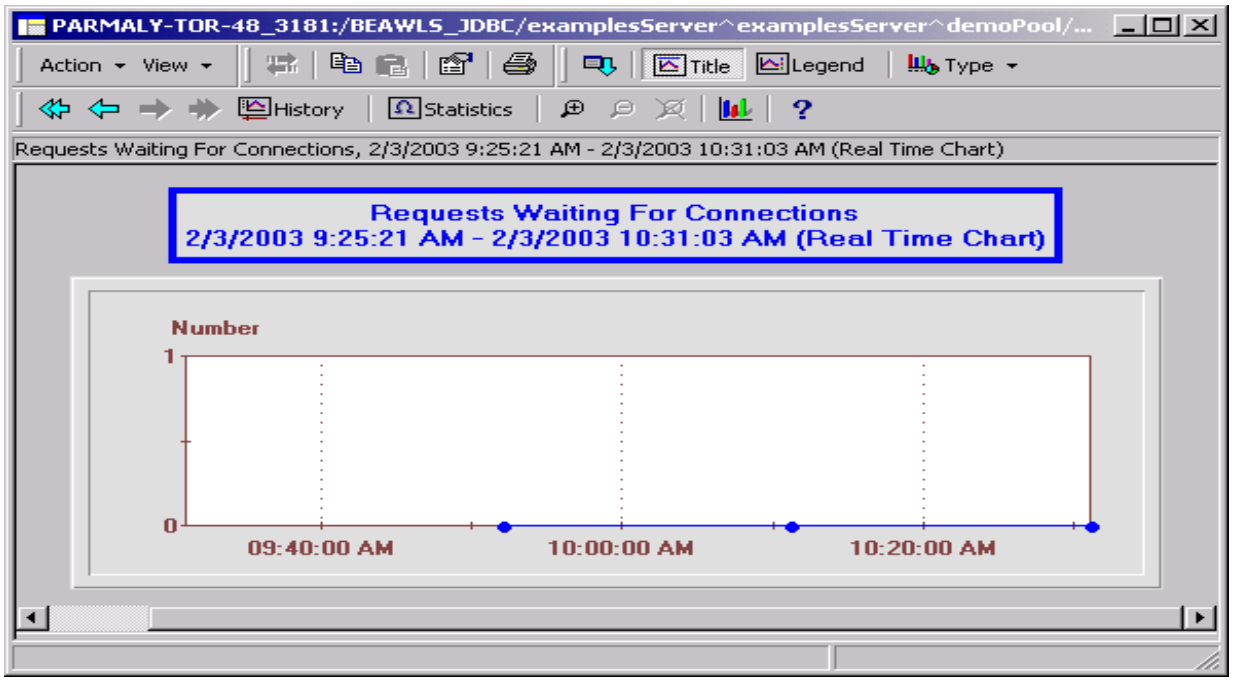
## Connection Pools

Java Database Connectivity (JDBC) is a standard Java API that allows developers to write database applications and execute SQL. Unfortunately, making JDBC connections with a database management system can be very slow. If an application requires multiple connections and they are repeatedly opened and closed, performance will be compromised. Connection Pools alleviate this problem by maintaining a pool of connections that are made when WebLogic Server is started. Physical connections are made and are available in the pool when applications request them. After they finish using the connection, the application will return the connection to the pool for reuse. By creating the connections at the startup of the pool WebLogic saves the overhead and the time it would normally take the application to make the physical connection on its own.

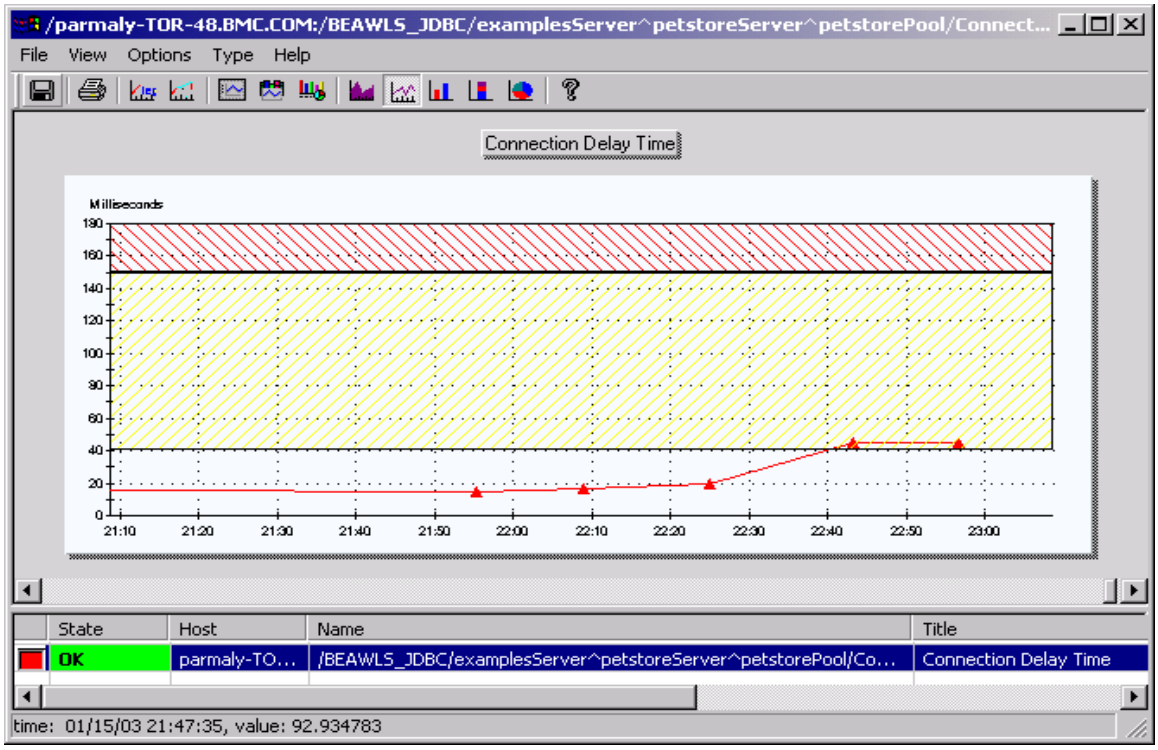
However, as with most other things concerning WebLogic, JDBC Connection Pools carve out a chunk of JVM memory (heap) to deliver the database connection speed the applications require. The challenge for administrators is how to accurately configure the size of the connection pools to balance the necessary throughput of the applications to the database with the amount of memory those pools are using within the JVM.

The graph below shows a critical JDBC parameter that an administrator should monitor. By examining the number of application requests that are waiting for a JDBC connection (i.e. waiting for another application to return the connection to the pool), the administrator can determine if the connection pools are properly configured. If the number of waiting requests grows then it most likely indicates the need to increase the size of the pool to handle the larger volume of application requests. The same reason applies for monitoring this parameter as the previous one. An application will be adversely affected if a thread cannot make a connection to the backend database. Connection pools are critical resources within WLS and careful consideration should be

taken to strike a balance between fast and instantaneous throughput and memory utilization (the larger the pool the larger the memory consumption).




What about less-obvious slowdowns, those that are not so readily recognized as the example just given? Consider the graph below. User activity forced a lot of requests for a connection to a database. There is normally a default login delay defined for each connection (can be configured through the WebLogic admin console or through Patrol) but if that delay grows it could indicate a problem with the database server or it could simply mean the administrator should increase the login delay to compensate for a very busy database server.

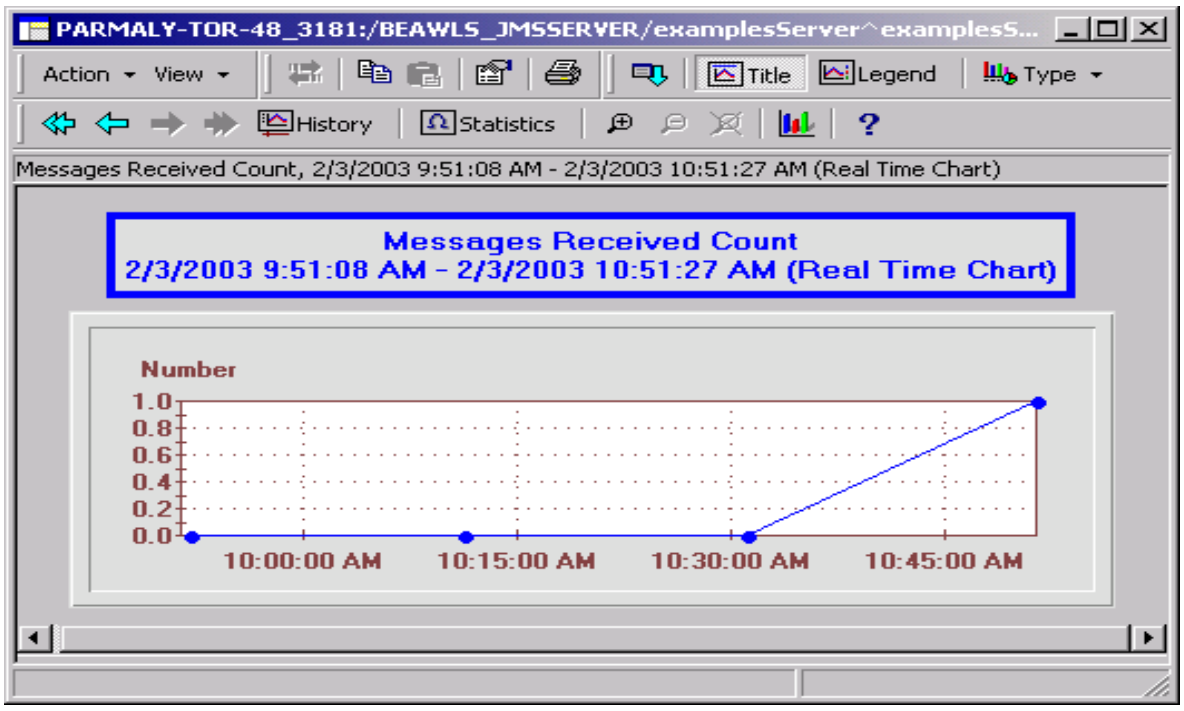



Being made aware of these types of conditions in advance of when a problem actually occurs is essentially the strength of Patrol (i.e. being notified before the connection delay is so high it slows the application down to the point where users complain or worse, give up and go elsewhere).

## JMS Server

Customers can use Patrol for BEA WebLogic to effectively monitor the Java Messaging Service (JMS) Server. This Java messaging component of WebLogic is a facility that facilitates communication between WebLogic components and also provides bridging connectivity for other messaging systems such as IBM's WebSphere MQ and Tibco's Enterprise for JMS. JMS is based on the idea of messages flowing to and through destinations (queues in the IBM MQ world). The message flow can be controlled to ensure prompt service and to limit the number of bytes and messages processed. Again, as with the other components of WebLogic previously discussed, the JMS Server uses memory within the JVM so care must be taken to ensure a balance is achieved.

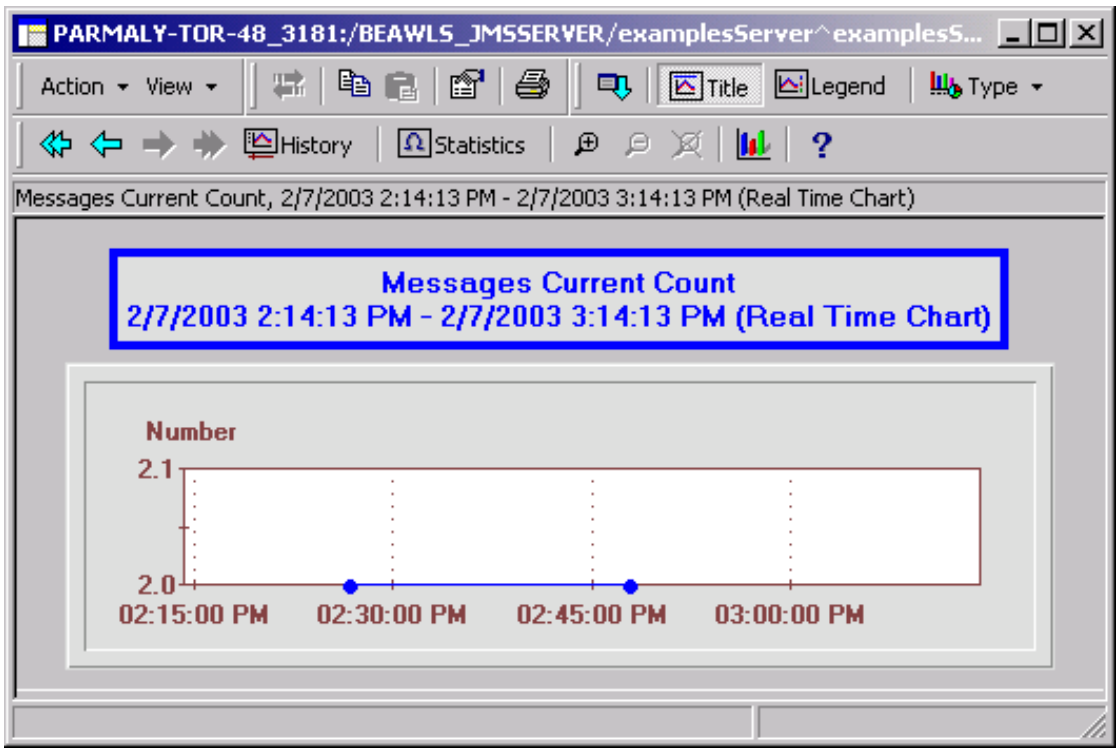
Patrol for BEA WebLogic has limited functionality with respect to JMS. The product monitors this component only has deep as the JMS Server level. It cannot monitor actual destinations, subscribers, and producers. Even so, Patrol can add value in situations where it is important to be forewarned that the JMS Server is busy and approaching the aggregate parameter settings.




The **Messages Received Count,** shown above, can be used as a method of being forewarned when the JMS Server is experiencing an unusual spike in activity. Even though this version of Patrol for BEA WebLogic Server does not show information down to the actual Destinations (queues) level a user of the product can still draw conclusions from the above graph. For example, if over a period of time at the same time each day, the number of messages received by the JMS Server reach a certain level and at the same time the JVM Heap size reaches a dangerously low level, then it should be safe to conclude that JMS traffic is causing the JMS Server to consume a lot of memory and is therefore affecting WLS performance. Remember, if the Heap size reaches a certain low threshold WLS will begin performing I/O to satisfy requests.

What can a Patrol for BEA WebLogic user do with this information? They could feel confident going into the WebLogic administrative console and establishing and configuring message flow control. Message Flow Control enables the JMS Server to tell producers of messages to slow down the flow of messages when a specific threshold has been reached. Conversely, it can tell producers to increase the flow when a specific threshold has been reached.

The graph below shows the number of messages currently in the JMS Server. By monitoring this parameter a Patrol for BEA WebLogic Server user can monitor how efficient the JMS Server has been configured to handle message load. A graph that shows a consistently high number of messages might indicate a need to configure a feature called Message Paging. That feature allows JMS Server to page out persistent messages to disk at times of heavy load, thus freeing up valuable memory.


PARMALY-TOR-48_3181:/BEAWLS_JMSSERVER/examplesServer^examplesS...
Action
View
Title
Legend
Type
History
Statistics
Messages Current Count, 2/7/2003 2:14:13 PM - 2/7/2003 3:14:13 PM (Real Time Chart)
Messages Current Count
2/7/2003 2:14:13 PM - 2/7/2003 3:14:13 PM (Real Time Chart)
Number
2.1
2.0
02:15:00 PM
02:30:00 PM
02:45:00 PM
03:00:00 PM

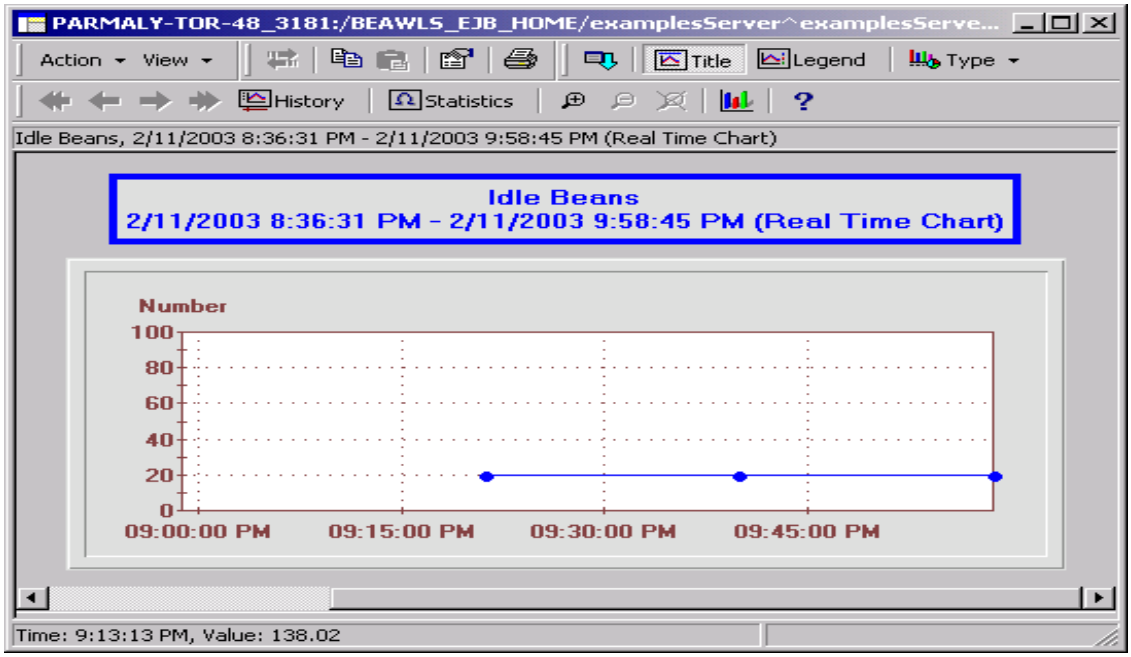
# Section II
# Other Performance Considerations

## Enterprise Java Beans

EJBs are fundamentally a Java specification that allows for reuse of services. Most commonly they act as bridges between the Web user and the back-end non-Web components that make up the application. The nature of an EJB is to be portable and as flexible as possible so that a developer, for example, need not necessarily include code to make the service transactional. Freeing the developer from concerns centered on transactional behavior, security, and lifecycle management allows them to concentrate on implementing and improving the business logic of the application. Making changes to applications that employ EJBs is much simpler and faster because the developer can make attribute changes such as altering the transactional behavior without touching the business logic.
That's the theory. Flexibility comes at a cost however. EJBs deployed with inefficient attributes can put a drag on the WebLogic Server and for that reason, monitoring their flow becomes an important performance goal.

## IdleBeanCount

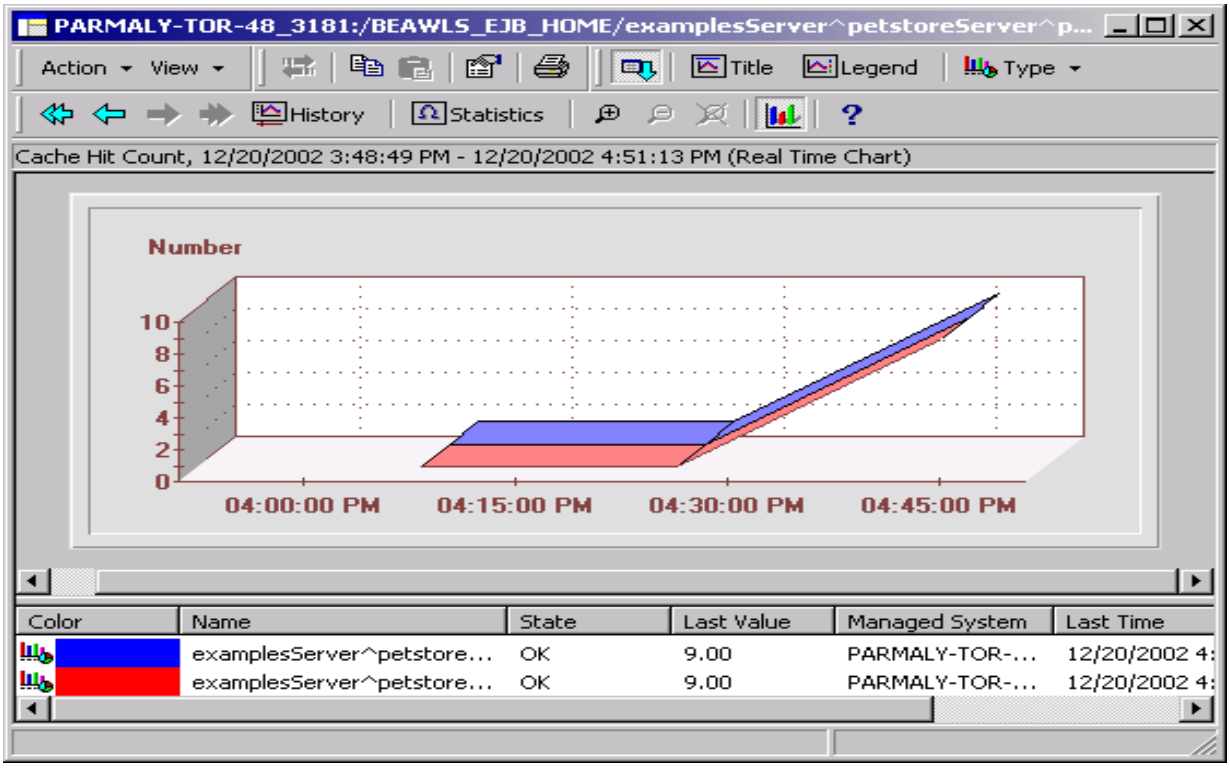
The graph below shows the number of EJBs currently idle within a pool that is managed by the Bean Container. The Bean Container provides services to EJBs, creating bean instances, managing pools of instances, and destroying them. It also handles threading, transaction support, data storage and retrieval for the beans. While a steady count of Idle Beans as shown in the graph is a normal event, a climbing graph would be a strong indicator that the container needs to destroy beans at a faster rate. A possible reason for an increased bean count is that a new application may have been deployed or an existing one modified so that the EJBs are stateful (contain session information across session restarts; a valid application method but not always necessary). This parameter should be monitored closely for anomalous conditions.




## Caching

Caching is one of the ways WebLogic Server improves response times for the application and the end user. It places user session information in memory in the form of EJBs. These session beans containing the user's session information are quickly and readily retrieved the next time the user visits that website. The administrator has control over how much session information is stored in the cache and needs to be mindful of striking a balance. From a user perspective it would be ideal if their session information was always found in the cache, thus ensuring a rapid response time when they visited the website. Isn't this the goal of IT managers? Yes it is but it is also their goal to optimize the use of WebLogic and ensure that the software is tuned to deliver the best service to all users. This means removing old session beans from the cache and to properly configure the cache size so that it does not consume too much memory of the JVM that WebLogic Server is running in.

In the graph below we can see two parameters that have been combined into a single display. I dragged the CacheAccessCount parameter into the graph showing the CacheHitCount. Together they provide a visual showing the success or failure rate of how frequently the applications found the desired object in the cache (i.e. 10 access attempts resulted in 10 successful retrievals).


PARMALY-TOR-48_3181:/BEAWLS_EJB_HOME/examplesServer^petstoreServer^p...
Action
View
Title
Legend
Type
History
Statistics
Cache Hit Count, 12/20/2002 3:48:49 PM - 12/20/2002 4:51:13 PM (Real Time Chart)
Number
10
8
6
4
2
0
04:00:00 PM
04:15:00 PM
04:30:00 PM
04:45:00 PM
Color
Name
State
Last Value
Managed System
Last Time
examplesServer^petstore...
OK
9.00
PARMALY-TOR-...
12/20/2002 4:
examplesServer^petstore...
OK
9.00
PARMALY-TOR-...
12/20/2002 4:

# Summary

The aim of this paper is to demonstrate that by focusing on the performance of a subset of WebLogic components, administrators of Patrol for BEA WebLogic Server can accelerate the time it takes to understand the architecture of BEA WebLogic Server to effectively tune the application server's performance. There are many more important BEA WebLogic configuration settings that are important and it is equally important to understand that Patrol for BEA WebLogic can monitor their behavior too.