

Tuning Techniques for Message Driven Beans

Robert Ryan | AQM Solutions, Inc.

May 2007

Since their introduction in the EJB 2.0 specification, Message Driven Beans (MDBs), have provided a means for asynchronously processing JMS messages for J2EE-based applications. With MDBs system designers are able to introduce architected background processing capabilities into their applications.

The MDB, via deployment descriptors and resource definitions, is associated with a particular JMS destination; when messages are sent to that destination, the J2EE container retrieves the message and delivers it to the associated MDB for processing.

The MDB, while an invaluable component for the system designer, can present a challenge to the performance analyst for several reasons. First, unlike HTTP-based transactions, MDB's do not typically have an end-user sitting at a desktop awaiting their response; second, because they are less visible, their impact to overall system performance is too often hidden. Finally, because the execution of the MDB execution is predicated by the presences of a message on a JMS destination (rather than a person physically striking an enter key), the number of discrete MDB executions can be wildly unpredictable. As integrated J2EE environments become more prevalent, the impact of poorly performing MDBs can spill over and adversely affect the responsiveness of the environment's HTTP-based workload.

Traditionally, the simplest approach to dealing with MDB performances has been to limit the amount of system resources available to the MDB. In order to correctly configure an MDB/JMS environment J2EE Administrators are required to define sets of related resources (such as message listeners, connection factories, and messages destinations) which contain sizing parameters for things like connection pools, session pools and thread pools. The basic tuning strategy employed by the J2EE Administrator's is to cap the amount of system resources allocated to MDB processing by artificially constraining the J2EE Application Server's ability to initiate too much concurrent MDB processing. (Note: for a more detailed discussions of this strategy I recommend Sanjay Kesavan's 2006 *Developwork's* article entitled "*How the maximum sessions property on the listener port affects WebSphere Application Server performance.*")

A second, complementary strategy, for minimizing the impact of MDBs that also exists is often overlooked by Administrators. Performing tuning at the application level can increase the efficiency of the actual MDB thereby allowing it to both get “more miles per gallon” out of existing resources and increasing overall system throughput. When contemplating competing design alternatives, application designers can usually understand which option will perform better, however, the ability to quantify potential performance differences can be considerably more difficult.

Toward this end AQM Solutions, tested some common MDB application configurations to order to quantify the performance differences under different design scenarios. Testing for each of the proposed alternative was performed on both a Z/OS (59 MIP Z9) and Window's (1.8 GHz Intel) environment. In the Z/OS environment, WebSphere Application Server 5.1 ran against an external WebSphere MQ 6.0 instance, while in the Windows configuration, WebSphere Application Server 5.1 was outfitted with IBM's embedded JMS provider.

Configuration 1: Processing JMS messages in batches

By default, WebSphere message listener ports are configured so that a separate transaction is wrapped around each message delivered to an MDB. However, the listener port definition can be modified to batch multiple messages within a single transaction boundary (see maximum messages setting). Recently, a large European telecommunications client was able to successfully tune their bill-generation processes by employing this exact technique. Our test was designed to measure the anticipated increase in throughput for “batched” MDB environments where two identical JMS queues were loaded with the same number of messages that were, in turn, delivered to identical MDBs. The only difference between the configurations were that the message listener for one of the queues was configured to wrap each message with a separate transaction while the other message listener was configured to allow up to 10 messages to be processed within the transaction. Message throughput was tracked by measuring the amount of time required for each MDB to process a fixed set of messages.

Results: The results indicated that by batching messages to an MDB, a 35% increase in throughput could be achieved in the Z/OS environment while a 65% increase was seen on the Windows platform. These results point to a clear performance win for applications that utilize this feature of the MDB listener.

Configuration 2: Non Persistent Queues

Performance problems can frequently arise out of the misapplication of a given facility. JMS allows messages to carry a delivery mode attribute that dictates whether messages are saved to a persistent storage (disk) prior to delivery. Persistent messages are generally saved while non-persistent ones are sent directly to their consuming MDB. There are many application scenarios in which an occasional lost JMS message will have little or no detrimental impact to the application. For example, an application deployed at a multi-national Insurance company uses MDBs to retrieve scanned images of historical policies for their claims processing division. Because the environment is extremely reliable stable, application designers decided to utilize non-persistent JMS messages thereby trading better overall throughput for the risk of a lost messages during rare system crashes.

To quantify the overhead differences between persistent and non-persistent JMS messages, a simple set of identical MDBs were created, one of which consumed persistent messages, while the other consumed non-persistent messages. The queues were then loaded with the same number of messages while their MDB listeners were active.

Results: Again, the measure for efficiency was the velocity in which the MDB container could drain the set of queue messages. As the MDBs were identical the throughput metrics effectively isolated the processing differences between persistent and non-persistent messages. Testing consistently showed that the non-persistent messages could be drained 15% faster on Z/OS and 20% faster on Windows.

Configuration 3: Use temporary JMS queues

A third test configuration investigated the usage of temporary JMS queues. A MDB design technique seen by AQM Solutions is one in which a "private" temporary JMS queue is used to provide synchronous-like responses to on-line applications. AQM recently encountered a call center application that provides near-time assembly of customer profiles. Inquiries entered by customer service representatives trigger background MDBs that develop the requested profile. The representative work-station locks for several seconds awaiting the response from the MDB. If the response is not timely, the work-station ceases waiting thereby allowing the customer service representative to move onto additional tasks.

Past experience on distributed platforms has indicated that the use of temporary queues significantly adds overhead and impacts application responsiveness on the order of 15% or more (this figure was confirmed

during our Windows environment). For this reason, it is often recommended that an MDB's reply queue be a shared, permanent queue rather a private, temporary one. While sharing a reply queue amongst simultaneous users adds the burden of sorting out which messages belong to which users, the performance profile of the shared permanent queue was always found to more attractive on the distributed environment.

Results: A simulation of these two techniques was developed for our test environment and was measured using a simple load-testing tool. Surprisingly, over the course of multiple test runs, there was no apparent performance penalty in the Z/OS environment (as measured by response time) for the temporary queue case. In fact, in all cases, the temporary queue case slightly outperformed the permanent case. Our hats go off to the IBM WebSphere designers and engineers as we now find ourselves endorsing the use of temporary queues in the Z/OS environment.

Configuration 4: Does the MDB really need to be transactional?

One of the really attractive features of J2EE is the container-managed transaction (CMT) that eliminates the need for the application to programmatically manage recovery scenarios. However, CMT comes with a price; the transaction envelope that is wrapped around the business logic can be the most expensive (resource-wise) component of the application processing. For applications that do not require the presence of a recoverable environments, bean managed transactions (BMT) can afford important performance benefits. When properly configured, the transaction envelope, and its associated overhead, can be completely eliminated. In the past, AQM worked with a large manufacturing concern that used MDBs to drive a sophisticated ad-hoc inventory reporting system. Because the MDBs associated with the triggering JMS messages were performing read-only inquiries against the manufacturing database, there was no need to incur the additional processing overhead associated with a container-managed transaction.

To test this scenario, a comparison of CMT and BMT MDBs was performed by creating two nearly identical MDB environments, differing only in that one MDB's deployment descriptor was configured for CMT while the other's was configured for BMT. The queues associated with these MDB were loaded with an identical set of messages and, as in previous tests; the rate in which the MDB's processed their messages was used as a basis for comparison.

Results: Message throughput was recorded by measuring the amount of time required for each MDB to process a fixed set of messages. Results indicated that BMT MDB's where able to achieve a throughput of about 2.5 times faster that of the CMT MDB (data from Windows/Intel environment only.)

Conclusion

Based on the results of the MDB benchmarks, it is evident that there are a variety of application design and configuration approaches that may be employed to increase the overall efficiency of an MDB-based application. The benefit of this study however is in the quantification of the resource consumption differences in competing alternatives. Because these techniques can be employed by modifying application configuration settings external to the application's business logic, they can be easily implemented once it was determined that an application's MDBs were, in fact, draining away important system resources. Obviously, any of the proposed approaches needs to be evaluated in light of one's particular application design.

This study should also highlight the requirement that individuals responsible for the performance of a J2EE application need to employ the necessary procedures, techniques and tooling to raise the visibility of the oft silent MDBs.

In most cases the reason for not employing many of these procedures is more logistical than any other. A busy administrator and his staff is often tasked with putting out day to day fires and has little time for identifying potential tuning opportunities. That is why many have chosen to outsource and eventually automate the process.

By optimizing the performance of your J2EE applications, you also optimize your business processes, and can play a significant role in improving enterprise wide efficiency and effectiveness. By measuring, interpreting and subsequently implementing tuning adjustments, small changes can save large amounts of money and greatly improve productivity.