BEA WebLogic™ JMS

Performance Guide

July 31, 2003

How Business Becomes E-Business™

This page intentionally left blank.

Table of Contents

1 Introduction

The *WebLogic JMS Performance Guide* explains how to tune BEA WebLogic Server™ JMS versions 6.0 through 8.1 to achieve better performance and higher availability. These goals are approached in two major sections: "Configuration Tuning" and "Application Design". Though targeted at WebLogic JMS versions 6.0 and later, some sections also apply to earlier WebLogic JMS versions, as well as to JMS implementations that are not based on WebLogic Server. This guide assumes a basic familiarity with WebLogic Server administration and JMS development.

Our main goal is to improve overall messaging performance, so we keep in mind that raw speed, though important, is only one of several performance-related factors. Other performance factors include reliability, scalability, manageability, monitoring, user transactions, message-driven bean support, and integration with an application server.

2 Related WebLogic JMS Information

For WebLogic Server 8.1, start with the BEA WebLogic JMS Topic Page.

For WebLogic Server 7.0 and earlier, see the BEA guides available on line:

- *BEA WebLogic Server Performance and Tuning*
- *BEA WebLogic Server Administration Guide* (specifically, "Managing JMS" and "Using the WebLogic Messaging Bridge")
- *Programming WebLogic JMS*
- *Programming WebLogic Enterprise JavaBeans* (for information on EJBs and MDBs)
- *Programming WebLogic XML*
- *Using WebLogic Server JMX Services* (configuration and monitoring MBean APIs)
- JMS white papers and FAQ located on dev2dev.bea.com (BEA's Developer Web-site) and on the BEA e-docs pages.

Javadoc for WebLogic Server JMX APIs and JMS extensions (links 8.1, 7.0):

- JMS configuration JMX MBeans, weblogic.management.configuration.JMS…
- JMS monitoring JMX MBeans, weblogic.management.runtime.JMS…
- JMS extensions, weblogic.jms.extensions…
- JMS ServerSessionPools, weblogic.jms.ServerSessionPoolFactory

BEA's Developer Web-site, dev2dev.bea.com, provides some useful JMX examples in its "Code Library/Code Samples" section. Some of the more popular samples are:

- `JMSStats.java`, a JMS statistics dump program
- *wlshell*, a comprehensive command-line configuration and monitoring tool
- *JMS Error Destination Manager*, a JSP-based tool for browsing and deleting queue messages (works with any queue, not just error queues)

The Java Message Service Specification and its Javadoc for the JMS 1.0.2 and 1.1 specifications.

> **Tip:** While BEA WebLogic Server has had JMS 1.1 capability since 7.0, the J2EE license forbids BEA, and other J2EE licensees, from officially supporting JMS 1.1 or advertising compatibility with it, until it is fully released. To enable JMS 1.1 capability with WebLogic JMS, download the JMS 1.1 interfaces and place them in your classpath.

BEA public newsgroups on newsgroups.bea.com, including:

- weblogic.developer.interest.ejb
- weblogic.developer.interest.jms
- weblogic.developer.interest.jndi
- weblogic.developer.interest.performance
- weblogic.developer.interest.rmi-iiop
- weblogic.developer.interest.transaction

Sample code bundled with WebLogic Server, including:

- examples/jms
- examples/ejb20/mdb
- examples/webservices/message
- http://dev2dev.bea.com/codelibrary/code/examples_jms.jsp

Recommended books:

- *J2EE Performance Testing with BEA WebLogic Server* by Peter Zadrozny, Philip Aston, and Ted Osborne.

  Tips: The examples in the JMS chapter of this book's first edition are universally "send limited", as only one JMS sender is used concurrently. This fundamentally limits scalability. This book makes heavy use of a java load-testing framework, "The Grinder", which is freely available under a BSD-style open-source license. Here is the link.

- *Mastering BEA WebLogic Server: Best Practices for Building and Deploying J2EE Applications* by Gregory Nyberg, Robert Patrick, Paul Bauerschmidt, Jeff McDaniel, and Raja Mukherjee. (Forthcoming August 18, 2003)

3 Benchmark Tips

To avoid misleading results when benchmarking a messaging application or messaging server, be sure to check the following aspects of performance:

Test Design/Benchmark Specials: Take care when evaluating competitive benchmarks, even when obtained through a neutral third party. For example, one tactic, used even by well-known vendors, is to deliberately benchmark narrow scenarios, while obscuring benchmark limitations via appealing graphs and tables. Neutral benchmarks suffer similarly, as they rarely match a particular application's performance characteristics well enough to be useful. Rather than using third party data and benchmarks, it is almost always best to model the target application(s) closely, a process that usually requires the development of a custom benchmark.

Throughput. Aggregate throughput (aggregate messages received per second) is usually the best measure of a messaging system's performance. Measuring latency is often misleading, as it fails to take into account scalability. Measuring messages sent per second is also often misleading, as the rate at which messages enter a system is usually not as interesting as the rate at which consumers actually handle the messages. Also, sends almost always use fewer resources than receives, and are consequently less likely to be a bottleneck.

Scalability. Can the server handle dozens, hundreds, or even thousands of clients? How is throughput affected as the number of clients increase? Ideally, it should increase as well.

Fairness. Ensure that homogenous applications tend to all get served at about the same rate, and that all take turns getting served. Be aware that some messaging products make absolutely no effort to promote "fairness" among consumers. This means that some consumers may end up getting serviced before others, and may even "starve" the others. In a queuing application, this could mean if there are several asynchronous consumers on the queue, some may rarely or never get messages. In a pub/sub application, this could mean a subscriber falls farther and farther behind the others. Or it could mean that the last subscriber to a topic is always the last to get a message – perhaps 1 second later than the first subscriber.

Dropped messages. WebLogic JMS does not "drop" (silently delete) messages, except where the designer has explicitly asked for a lower quality of service (such as using the multicast option for pub/sub, see 5.7, or message expiration, see 5.15). This is not true for all vendors – as the JMS specification allows a lower quality-of-service for pub/sub messages than one might expect; it allows messages bound for non-durable subscriptions to be arbitrarily dropped. Some vendors drop pub/sub messages in case of congestion. One vendor drops pub/sub messages for synchronous non-durable subscribers if the subscriber doesn't currently have a blocking receive posted on the server. Not detecting dropped messages when comparing messaging vendors can lead to misleading results.

Apples-to-apples. Take care that competing vendors are run on the same hardware and are run with the best JVM available for the vendor. If possible, use the same JVM for both.

Concurrency. It is sometimes useful to measure whether code runs in parallel when expected. Like *fairness*, this can yield surprising results from application to application and from vendor to vendor. For example, JMS asynchronous consumers that share a JMS session must not run concurrently. If they do run concurrently, then the JMS vendor is violating the JMS specification, which requires that sessions be single threaded. Violating this aspect of the JMS specification yields a "benchmark special". To get concurrency, an application must use multiple sessions.

Persistence. How is the message store implemented, via JDBC or a file store? WebLogic JMS supplies both. If it is a file store, are synchronous writes used for both sends and receives? Some JMS vendors do not synchronize disk writes by default, and so will drop and/or duplicate messages on a power failure. Failure to synchronize disk writes makes for deceptively high benchmark numbers. WebLogic JMS provides an option to turn off synchronous writes for applications that don't need this quality of service (see 4.4.3.1). This leads to the next question, what happens if a computer crashes or a disk gets corrupted? At least two JMS vendors fail to detect if their message stores are corrupted. These vendors therefore improve performance, but at the expense of unpredictable behavior.

Sampling. Be careful to measure performance over a statistically significant number of messages. Using a small number can result in large variations from test to test, and can fail to give JVMs that perform dynamic optimizations time to do their magic (such as the BEA JRockit 8.1 JVM). One way to get more reliable numbers is to steadily increase sample size until the results are repeatable with a low standard deviation. Another is to steadily monitor a long running test until a throughput moving average varies little as time goes on.

Consistent performance. What happens when the messaging server and/or application gets pushed very hard? Does the messaging server or application provide throttling to handle stress cases or short-term bursts of traffic? Or do they keep accepting more work until a crash? To help ensure consistent performance see section 5.18 on *Overflow Conditions and Throttling*.

Network bottlenecks. Network bandwidth can quickly become a limiting factor for high data rates or for slow networks. For additional information, see section 5.17.

Application bottlenecks. The cost of messaging is often not a significant fraction of an application's workload. The application is often doing much more than just messaging (e.g. database updates), or the application itself is not heavily used. This guide may still help, as it addresses concepts that apply beyond pure messaging, but consult the documentation for your other subsystems for further information.

Hardware bottlenecks. When benchmarking, it is often useful to simultaneously sample hardware statistics such as CPU percentage and disk usage. Note that disk usage statistics are sometimes misleading, as a disk that is 100% utilized may still support more application work due to aggregation of multiple concurrent I/Os into one.

Benchmark hardware. Take care that benchmarks model real world hardware configurations. For example, a benchmark that puts all clients and servers on the same machine will often yield results that don't even come close to modeling an actual distributed application.

Benchmark overhead. Ensure that any instrumentation and/or benchmark code is lightweight. There exist benchmarks that call "System.out", or make network calls, or log to disk so frequently that the benchmark itself steals significant resources from the actual application.

4 Configuration Tuning

This section provides performance guidelines for JVM and WebLogic Server configuration.

4.1 Tuning Thread Pools and EJB Pools

Thread pool sizes on the client and server, and similarly for EJB pools, are tunable.

Tuning thread pool sizes either up or down may have a positive performance impact.

Increasing thread pool sizes may increase performance in the case where many threads block while waiting for external work to complete. On the other hand, increasing thread pool sizes may substantially *decrease* performance as it can increase "context-switching" overhead and increases heap occupancy. Context switching is the cost that is incurred when an operating system switches a CPU from one thread to another.

4.1.1 Surveying the Current Threads

You can get an idea of how many threads are in use by forcing a thread dump, which enumerates all of the threads as well as their method stacks. Here are some ways to get a thread dump:

- On Windows, simultaneously press the "Ctrl" and "Break" keys on the java console window. This works on a server as well as a client.

- On Windows, if the goal is simply to count the total number of threads, simply use the "Windows Task Manger" (ctrl-alt-delete), go to the "Processes" tab, select the "View" menu, click "Select Columns" and then check "Thread Count". Alternatively, you can use the Windows "perfmon" utility, add the process as performance object, select the java instance (WLS), and use the thread count counter.

- On Unix, signal the process to dump using "`kill -3 <process-id>`". This works on a server as well as a client.

- On any operating system, another way to view a server's threads is through the WebLogic administrator console. In 8.1 and 7.0, this is available using the "Servers -> <server name> -> Monitoring Tab -> "Monitor all Active Queues" link. The queues being monitored here are thread pool queues (not JMS queues). The same values displayed on the console are available through a JMX MBean API "ExecuteQueueRuntimeMBean"; see the Programming WebLogic Management Services with JMX guide for details on how to access the JMX API.

4.1.2 Client-Side Thread Pools

WebLogic client thread pools are configured differently than WebLogic server thread-pools. Weblogic clients have a thread pool that is used for handling incoming requests from the server, such as JMS MessageListener invocations. This pool, called the "execute queue", has a default size of 5 threads. It can be configured in 8.1, 7.0, 6.0, and 5.1 via the command-line property:

```
-Dweblogic.ThreadPoolSize="n"
```

and in 6.1 via the command-line property:

```
-Dweblogic.executequeue.ThreadCount="n"
```

With most applications, using the default execute queue size is sufficient. If, however, the application has a large number of consumers, then it is often beneficial to allocate slightly more execute queue threads than consumers. This allows more consumers to run concurrently.

You can also configure a JMS specific thread pool (default size 0) on the client via a command-line property, *but tests have revealed no measurable benefit in tuning this value, so the reader may want to skip to the next section.* The command-line property is:

```
-Dweblogic.JMSThreadPoolSize=n
```

where `n` is a positive integer. This setting enforces a minimum of 5, so if you set it at all you get at least 5 threads. Once set, incoming JMS requests are dispatched to the "JmsDispatcher" thread pool instead of the execute queue thread pool. Additional executes (work that cannot be completed in the initial request thread) are executed in the "default" execute queue.

4.1.3 Server-Side Thread Pools

On the server, incoming JMS related requests execute in the JMS execute queue/thread pool. Additional work that cannot be completed in the request thread is forwarded to the "default" execute queue. In addition messaging bridges running in synchronous mode also use their own thread pool. The server-side default execute queue thread pool and JMS thread pool sizes can be set via command-line properties as described in section 4.1.2, but it is generally easier to configure them instead. A sample 8.1 and 7.0 configuration:

```
<Server Name="server1"
   JMSThreadPoolsSize=”8”
   MessagingBridgeThreadPoolSize=”1”
   >
   <ExecuteQueue Name="default" ThreadCount="8"/>
</Server>
```

Alternatively, the JMS thread pool size can be configured directly as an execute queue, which has advantages in that it more easily monitored on the console:

```
<Server Name="server1"
   >
   <ExecuteQueue Name="default" ThreadCount="8"/>
   <ExecuteQueue Name="JMSDispatcher" ThreadCount="8"/>
   <ExecuteQueue Name="MessagingBridge" ThreadCount="1"/>
</Server>
```

The equivalent 6.1 configuration:

```
<Server Name="server1"
   ThreadPoolSize="8"
   JMSThreadPoolSize=”8”
   >
</Server>
```

The default size for the server-side JMS pool is 15 threads and for the default execute queue it is also 15 threads. The JMS pool minimum size is 5 – no matter what it is set to. The JMS pool size can be configured on the console under “Server -> <server-name> -> Services -> JMS Thread Pool Size”.

For information on the messaging bridge thread pool see section 4.9.5.

For more information, see “Using Execute Queues to Control Thread Usage” in *WebLogic Server Performance and Tuning* (links 7.0, 8.1).

4.1.4 Server-Side Application Pools

Server-side applications get their threads from the service in which they are running. This service may be RMI, Servlets, EJBs, or MDBs. When a server-side application calls into WebLogic JMS, WebLogic JMS uses this thread as long as possible. Work that cannot be completed immediately by the application thread is scheduled on the "default" execute queue and the application thread then blocks (in the same way it blocks for a request to a remote JVM). When the work is completed, the blocking thread is resumed.

Depending on your application, you may want to restrict application concurrency to a smaller number of threads than the total available number for a variety of reasons:

- *Semi-Starvation:* By continuously using all threads, a very active server-side application may slow down other higher priority applications.

- *Deadlock*: A server-side application may introduce deadlock situations where an application waits for a response indefinitely because the responder is blocked while waiting to get a thread.

- *Performance*: A server-side application may use more concurrent threads than it needs. This slows down performance, as it increases CPU context switching.

Application thread usage and concurrency may be tuned through the following methods:

- *Performance*: A server-side application may use more concurrent threads than it needs. This slows down performance, as it increases CPU context switching.

- *For EJBs and MDBs:* Configuring the application's `max-beans-in-free-pool` element in its `weblogic-ejb-jar.xml` descriptor file to a size that is smaller than the size of thread pool it shares. The name `max-beans-in-free-pool` is somewhat of a misnomer; it is more accurate to consider this element a "maximum concurrency" parameter.

  Tip: As of WebLogic 8.1, the `max-beans-in-free-pool` descriptor attribute, as well as the message selector, JMS polling interval, and transaction timeout attributes are dynamically configurable at runtime via the console via Deployments -> Applications -> <application name> -> <application jar>. To enable dynamic update of the MDB descriptor, deploy the EJB module in exploded form.

- *For EJBs, RMI, and Servlets*: Assign a dedicated thread pool to the application.

  For EJBs and MDBs the dedicated thread pool can be set via the dispatch-policy value in the weblogic-ejb-jar.xml:

  ```
  <weblogic-enterprise-bean>
    <ejb-name>hi</ejb-name>
    ...
    <dispatch-policy>CriticalAppQueue</dispatch-policy>
  </weblogic-enterprise-bean>
  ```

- *For MDBs:* As with EJBs above, MDBs may also be assigned a dedicated thread pool using the `dispatch-policy` attribute. This capability was added for MDBs in 8.1, and 7.0SP3 and later; in previous versions, the `dispatch-policy` attribute can be configured but is ignored.

  If an MDB's dispatch policy is not configured, is not supported, or refers to an unknown thread pool, MDBs use their server's default execute queue.

  If an MDB is executing in the default execute thread pool, and the MDB is running on a WebLogic version 6.1SP5, or versions 7.0 or later server, an MDB deployment limits its maximum concurrent instances to half of the configured size of the default execute queue plus one. This self limiting behavior helps prevent a slow MDB from consuming the entire default execute thread pool, which, in turn, helps prevent dead-locks and possible starvation of other resources that use the same thread pool.

For more information see:

- "Tuning WebLogic Server EJBs" in *BEA WebLogic Server Performance and Tuning*. Links 7.0, 8.1.
- "Using Execute Queues to Control Thread Usage" in *BEA WebLogic Server Performance and Tuning*. Links 7.0, 8.1.
- dispatch-policy in *weblogic-ejb-jar.xml Deployment Descriptor Reference*
- wl-dispatch-policy in *weblogic.xml Deployment Descriptor Elements*
- Section 4.1.3 for information on how to configure the default execute queue.

4.2 Network Socket Tuning

4.2.1 Tuning Server Sockets

By default, a WebLogic Server uses native socket drivers. Native socket I/O is enabled via the server's NativeIOEnabled flag, and greatly enhances the ability of a server to handle multiple remote clients while using few threads. If native socket drivers are disabled, you will likely need to increase the number of default execute threads (see section 4.1.3) and tune the server's "ThreadPoolPercentSocketReaders" setting. Otherwise, performance may degrade by as much as 90%. If native threads are disabled, allot one thread per remote JVM unless a mix of SSL and plain-text sockets are used, in which case allot two threads per remote JVM. For more information, see "Using WebLogic Server Performance Packs" and "Allocating Threads to Use as Socket Readers" in *BEA WebLogic Server Performance and Tuning*.

Note: When native socket drivers are enabled, the "ThreadPoolPercentSocketReaders" setting is ignored.

4.2.2 Tuning Client Sockets (WebLogic 6.1SP1 and earlier)

WebLogic clients do not use the native socket drivers. If a WebLogic client makes connections to multiple WebLogic servers (more than two or three), it may suffer serious performance degradation unless additional threads are allocated for the sockets. WebLogic 6.1SP2 and later clients automatically allocate threads to handle additional sockets, but earlier clients do not. To configure additional threads for the client, see section 4.1.2. To increase the percentage of threads allocated to Java sockets, set the following command-line property:

```
-Dweblogic.ThreadPoolPercentSocketReaders=50
```

Allot one thread per remote server JVM unless a mix of SSL and plain-text sockets are used, in which case you may need to allot two threads per remote server JVM.

Note: WebLogic multiplexes all network traffic from one JVM to another over a single socket, no matter how many contexts and JMS connections exist between the JVMs.

4.2.3 Tuning WebLogic Network Packets (Chunks)

4.2.3.1 Tuning Chunk Size (Advanced Use Only)

WebLogic JVMs divide network data into 4K chunks by default. For network intensive applications, tuning the chunk size can sometimes *double* throughput. The chunk size is set via a command-line system property on both server *and* client:

```
-Dweblogic.Chunksize=n
```

Chunk size tuning can be useful to accommodate workloads with particular payload sizes. It may improve performance to set the chunk size smaller for high-frequency small payload workloads, or larger for big payload workloads. Chunk size tuning is also useful for matching network MTU size and operating system memory page size.

Restrictions: The chunk size must be a multiple of 8. In addition, the chunk size interacts with the MTU of the network being used. The chunk size should be a multiple of the MTU size, and in no circumstances should the remainder be a small fraction of the MTU. This will result in significant fragmentation of the packets being sent and a corresponding loss of performance. The following values are recommended for experimentation (the 16 fewer bytes accounts for internal WebLogic internal headers that are appended to the chunk):

```
2032, 4080, …, (4096*n)-16
```

Note: The default chunk size is 4080. Operating systems often allocate their memory pages in 4K boundaries, so the default chunk size of 4080 is compromise between MTU size and operating system page size.

Note: For asynchronous consumers, WebLogic JMS aggregates (pipelines) multiple messages into one payload. In effect, this increases the network size of messages pushed to asynchronous consumers. For more information on the asynchronous pipeline see section 4.8.

Note: In eGenera systems, the optimal value for the chunk size is 16K or 32K in order to match the eGenera MTU size. This should be set on both the server and client JVM's. For more information, see http://e-docs.bea.com/wls/certifications/certs_700/eGenera.html.

4.2.3.2 Tuning Chunk Pool Size

WebLogic caches network buffers (chunks) in an internal chunk pool for re-use. The size of the pool is set via a command-line system property on a server and/or client:

```
-Dweblogic.utils.io.chunkpoolsize=n
```

This can be useful to accommodate workloads with particular payload sizes or concurrent client activity. Setting it smaller might be useful for small payload workloads, larger for big payload workloads. Setting it larger can also be useful to accommodate workloads with many concurrent clients. The default chunk pool size is 512.

Restrictions: The chunk pool size is tunable in 6.1SP5, and versions 7.0 and up.

4.3 JVM Tuning

On a Sun JVM, you can improve performance by setting the minimum and maximum amounts of heap memory to the same value. To configure these parameters for version 1.3, specify the `-Xms` and `-Xmx` options on the `java` command line. For example, to reserve 64 MB of heap:

```
java -Xms64m -Xmx64m
```

For JMS applications that have many concurrent non-persistent clients, WebLogic Servers running on BEA JRockit JVMs may need tuning to reduce lock contention. When encountering a contended lock, the following option enables the JRockit JVM to "spin" for a short time retrying an inexpensive "thin lock" before using an expensive "fat lock".

```
-XXenablefatspin
```

The "`enablefatspin`" option is documented under "Dealing with Heavily Contended Locks" in *Release Notes for Version 8.1 Service Pack 1*.

There are numerous other JVM tuning options. For details about JVM tuning, see "Tuning Java Virtual Machines (JVMs)" in *BEA WebLogic Server Performance and Tuning*. For information on BEA's jRockit JVM tuning see "Tuning WebLogic JRockit 8.1 JVM" in the jRockit Documentation.

4.4 Persistent Stores

4.4.1 JDBC Stores vs. File Stores

WebLogic JMS 6.0 and later can store persistent messages in either a database or a file. The following are some similarities and differences between file stores and JDBC stores.

- File stores and JDBC stores both have the same transaction semantics and guarantees. As with JDBC store writes, file store writes are guaranteed to persist synchronously to disk and are not simply left in an intermediate (unsafe) cache.
- File stores and JDBC stores both have the same application interface (no difference in application code).
- File stores are generally faster. Sometimes much faster.
- File stores are easier to configure.
- File stores generate no network traffic; JDBC stores will generate network traffic if the database is on a different machine from the JMS server.
- File stores are much better suited to paging non-persistent messages (for more on paging see 4.7).
- JDBC stores may make it easier to handle failure recovery since the JMS store can access the database data from any machine on the network; with file store data, the disk must be shared or migrated when the JMS server is moved to another machine.
- JDBC stores can take advantage of high performance disk hardware (such as a RAID) that is often already available for database servers.
- JDBC stores are sometimes easier to manage because they centralize data on the database server. No need to manage each managed server's physical disks separately.

4.4.2 JDBC Store Tuning

4.4.2.1 Boot Performance

When a JMS JDBC store is used, WebLogic Server JMS may spend a significant amount of time scanning all database tables to find its own table**.** If, however, the name of the store includes a unique prefix that includes the schema name, this search time can be reduced, thus improving boot performance.

For this reason we recommend adding a unique prefix to the store name when configuring a JMS JDBC store. The prefix may be any string, but in many databases, the user name is used as the schema name.

Use the following syntax for the prefix:

```
[[[catalog.]schema.]prefix]
```

The periods in the prefix are significant.

4.4.2.2 Multiple Database Stores

Distributing a destination across multiple JMS servers, and therefore multiple database stores, may quite significantly improve performance, even when all of the database stores use the same database. If it appears that a single destination has reached its maximum performance at less than the maximum number of concurrent clients, this option is well worth investigating, otherwise, performance may actually degrade as the number of stores is increased. For more information on distributed destinations see section 4.6.1.1.

Tip: Multiple JMS servers, and therefore multiple stores, may run concurrently on the same WebLogic Server.

4.4.2.3 Oracle Database Tuning

This section provides tuning tips for configurations in which Oracle is used as a backing store for JMS JDBC stores. Some of the parameters mentioned here apply to other databases, but will have different names. Because some settings produce side effects, it is important to consult the Oracle documentation before changing any of the settings discussed in this section.

1) Increase DB_BLOCK_BUFFERS in SGA (shared memory global area)
The DB_BLOCK_BUFFFERS parameter specifies the number of Oracle blocks to reserve in memory for caching indexes and tables.

2) Increase SHARED_POOL_SIZE in SGA (shared memory global area)
The SHARED_POOL_SIZE parameter specifies the space reserved for caching queries and the dictionary (including the schema). Note that increasing this too high can actually degrade performance.

3) Net8 packet size (SDU)
Determine how to set this parameter on the basis of the average size of a message handled by your application. The default size, 2K, may be too small. This setting may not be configurable for type IV drivers.

4) Rebuild indexes once a month or so
Without periodic rebuilds, indexes may become inefficient and slow down table access.

5) Redo file size
To decrease the number of checkpoint operations performed by the database, increase "redo file" size.

You may also have to set log_checkpoint_interval high and log_checkpoint_timeout=0 (to turn off checkpoint based on timeouts). Note that this tuning potentially increases database recovery time after a database failure.

6) Oracle Extent Size INITIAL and NEXT
To increase the amount of contiguous space initially allocated to a table, increase the value of INITIAL. If you know, beforehand, the amount of space that will be used by your application, use this amount as the value of INITIAL.

To increase the size by which extents are increased when the table expands, increase the value of NEXT.

Note: Some DBAs recommend not using INITIAL and NEXT in favor of using Oracle's "locally managed tablespaces". With "locally managed tablespaces", Oracle uses a different and presumably better mechanism for storing extent meta-data,

7) Oracle optimizer mode
Because the SQL queries used are simple (for example, they do not perform joins), tuning this parameter may not be helpful for applications built on JMS version 6.0 and later. On the other hand, fairly complex queries, including embedded queries and multi-way joins, are used in JMS 5.1. The default *cost-based* mode is generally recommended, but *rule-based* mode may be faster for JMS 5.1 – particularly for older versions of Oracle.

8) Turn off table statistics
Administrators often turn on table statistics to get DB statistics, but doing so can hurt performance. To turn off table statistics, issue the appropriate command. The syntax for this command should appear, roughly, as follows:

```
analyze table <tablename> delete statistics
```

9) The _bump_highwater_mark_count parameter.
To insert into a table, a process must obtain blocks from a freelist. To obtain a freelist, the process must obtain an "HW enqueue". In insert intensive applications (such as WebLogic JMS), the "HW enqueue" can become a bottleneck. For one application, we found it was beneficial to increase the _bump_highwater_mark_count from 5 to 485.

Note: Similarly, it may also be helpful to increase the number of freelists.

10) The _spin_count parameter.
As the number of users accessing Oracle rises, thus increasing contention on the shared pool latch, response time also rises. CPU usage becomes sporadic as Oracle processes rush to obtain the shared pool latch, only to end up spinning and going to sleep. Increasing the _spin_count init.ora parameter can help. Use the _spin_count init.ora parameter to control the number of times an Oracle process spins for a latch before going to sleep.

This parameter is notorious for being set too low. In Oracle 6.0, the default value of spin_count was set to 2,000 and it has never been changed, despite the fact that CPU performance has since increased, effectively reducing the time an Oracle process waits for the latch before going to sleep. Measurable performance benefits have been observed when the spin_count is increased to 10,000.

11) Split Oracle across several physical hard drives:
2 for "redo" (If multiplexing is used, set redo to 4.)
1 per archive (Needed for up-to-second recovery; turned on via "Archive Log Mode".)
1 per table
1 per index
1 for the O/S
1 for the Oracle binaries

12) Consider configuring Oracle to use raw disk devices:
Oracle can be configured to use raw disk partitions for data storage, which, in some cases, can significantly benefit Oracle performance.

4.4.3 File Store Tuning

4.4.3.1 Disabling Synchronous Writes

By default, WebLogic JMS file stores guarantee up-to-the-message integrity by using synchronous writes. WebLogic JMS provides an option to disable synchronous writes. *Disabling synchronous writes improves file store performance, often quite dramatically, but at the expense of possibly losing sent messages or generating duplicate received messages if the operating system crashes or a hardware failure occurs.* Such failures are not caused simply by the shutdown of an operating system, because an operating system flushes all outstanding writes during a normal shutdown. Instead, these failures can be emulated by shutting the power off to a busy server. Unlike WebLogic JMS, some JMS vendors disable synchronous writes by default, and one JMS vendor only allows enabling synchronous writes for sends (not for receives).

To disable synchronous writes for a WebLogic 7.0SP1 and later JMS file stores, use the console to set *JMS stores -> "Synchronous Write Policy"* to "Disabled", or add the following attribute to the file store's config.xml entry:

```
SynchronousWritePolicy="Disabled"
```

For WebLogic versions earlier than 7.0SP1, disable synchronous writes for all file stores on a WebLogic server using the command-line property:

```
-Dweblogic.JMSFileStore.SynchronousWritesEnabled=false
```

and disable synchronous writes for a particular JMS file store using:

```
-Dweblogic.JMSFileStore.store-name.SynchronousWritesEnabled=false
```

If both properties are set, the latter overrides the former

A log message is generated when synchronous writes are disabled. This message can be used to verify that a command-line property is taking effect.

Note: This setting has no effect on file based non-persistent paging stores (see section 4.7).

4.4.3.2 Enabling Direct File Writes

WebLogic 7.0SP1 and later JMS file stores may optionally use direct writes. Direct writes may significantly improve performance when there are few concurrently active JMS clients, or when there a many concurrent JMS clients and the hardware disk write cache is safely activated, but may otherwise degrade performance. To enable direct file writes for JMS file stores, use the console to set *JMS stores -> "Synchronous Write Policy"* to "Direct-Write", or add the following attribute to the file store's config.xml entry:

```
SynchronousWritePolicy="Direct-Write"
```

Except for certain cases on Windows, this setting has the same transactional guarantee as the default "Cache-Flush" setting (see console documentation and see below for details).

WebLogic 7.0SP1 and later support direct file writes on Windows and Solaris. WebLogic 7.0SP3 additionally supports HP. If you would like support on another platform, contact BEA customer support.

Tip: As of WebLogic 7.0SP1, direct file writes may also be enabled for the WebLogic server's transaction log. For more information see the console: Servers -> <server name> -> Logging Tab -> JTA tab -> Transaction Log File Write Policy.

Tip: This setting has no effect on file based non-persistent paging stores (see section 4.7).

IMPORTANT WINDOWS NOTE: ***As with any product that uses direct writes, care must be taken when using the default Windows disk configuration. The default Windows disk configuration may be high performance but transactionally unsafe. It is important to understand that most "C" based products use direct writes (for example, MQSeries, Tibco, and Oracle).***

On Windows 2000 and XP, Direct-Writes takes advantage of the "Disk Write Cache", and perform significantly better when "Disk Write Cache" is enabled (the default). If Windows' "Disk Write Cache" is enabled, the hard disk write-back cache gets activated which, as documented by Microsoft, may lead to file system damage or data loss in the power failure, "improper" shutdown, or operating system crash.

Direct-Writes using Windows "Disk Write Cache" enabled is recommended only if the hard disk's write cache is battery backed up. As part of a fix to XP and 2000, Microsoft has recently added a

new flag "Power Protected" in addition to "Write Caching" for disk configuration. "Power Protected" is used to tell Windows that the on-disk write cache is battery backed, so that it need not force a cache flush down to disk when a native write occurred.

The following articles from Microsoft's knowledge base describe the above disk configuration features and the potential of data loss:

> *Description of Advanced Disk Properties Features*
> *http://support.microsoft.com/default.aspx?scid=KB;en-us;q233541*
>
> *Possible Data Loss After You Enable the "Write Cache Enabled" Feature*
> *http://support.microsoft.com/default.aspx?scid=kb;EN-US;281672*
>
> *Slow Disk Performance When Write Caching Is Enabled*
> *http://support.microsoft.com/default.aspx?scid=kb;[LN];332023*

4.4.3.3 Multiple File Stores

As with database stores, distributing a destination across multiple JMS servers, and therefore multiple database stores, may significantly improve performance, as long as all of the stores are not all placed on a single disk. If it appears that a single destination has reached its maximum performance at less than the maximum number of concurrent clients, this option is well worth investigating, otherwise, performance may actually degrade as the number of stores is increased. For more information on distributed destinations see section 4.6.1.1.

Tip: Multiple JMS servers, and therefore multiple stores, may run concurrently on the same WebLogic Server.

Tip: Multiple file stores benefits RAID configurations where disks are striped (see next section).

4.4.3.4 Enhanced Hardware and Multiple File Stores

For improved performance, consider the following:

- Distributing destinations among different JMS servers, each with its own store, and placing the stores on separate disks or in multi-disk RAID storage. Note that more than one JMS server may reside on a WebLogic server. Be careful, however, as partitioning destinations in this way can actually slow down some applications: see section 4.5.1.

- Dedicating a separate disk for JMS stores to avoid sharing the disks with the WebLogic Server transaction log or other applications.

- Using high-performance hardware, for instance hardware that supports disk striping.

For improved reliability, consider using hardware disk mirroring.

For fail-over capability, consider using dual-ported disks or a storage-area-network (SAN), which allow multiple hosts to share the same disk.

4.5 Partitioning Your Application

4.5.1 Multiple Destinations Per JMS Server

It is often best to configure multiple destinations to reside on a single JMS server, unless scalability seems to have peaked, in which case consider spreading different destinations among multiple JMS servers. Multiple JMS servers may still optionally be targeted at the same WebLogic server instance, but they will not share their persistent store.

Note: Transactions that span more than one JMS server become two-phase, which impacts performance. For more information, see section 4.5.4

4.5.2 Using the WebLogic Messaging Bridge

Tip: For information about tuning the Messaging Bridge, see section 4.9**.**

Using the WebLogic Messaging Bridge, you can partition an application between different sites, or even between different JMS vendors. The Messaging Bridge is capable of transferring messages from destination to destination between any two JMS-compliant vendors, and/or between any two versions of WebLogic JMS. For example, a Messaging Bridge running on a WebLogic 8.1 or 7.0 server instance can be used to forward messages from WebLogic 5.1 queue into a WebLogic 6.1 topic. If a source or target destination is unavailable, the bridge automatically makes repeated attempts to reconnect until the source or target destination becomes available again.

Why does a messaging bridge belong in a performance document? First of all, a messaging bridge can be used to replicate a topic, similar to the distributed topics feature available in versions 8.1 and 7.0, consequently improving scalability and high availability in some scenarios. Topic replication is accomplished by configuring the bridge to subscribe to one topic and forward the topic's messages to another topic, in essence creating two topics with the same message stream. For more information see sections 4.6.1.1 and 7.

Secondly, a messaging bridge can be used to effect high availability of remote destinations (otherwise known as "store and forward"). It enables local clients to produce to a local destination and have those messages automatically forwarded to the remote destination when it is available. Therefore, local clients can continue to produce messages even when the remote destination is currently unreachable.

The WebLogic Messaging Bridge is available on WebLogic versions 7.0 and later, as well as 6.1SP3. For 7.0 information about configuring the Messaging Bridge, refer to "Using the WebLogic Messaging Bridge" in *BEA WebLogic Server Administration Guide.* For 8.1 information, refer to "Messaging Bridge" in *Administration Console Online Help*.

4.5.3 Co-locating JMS with Server-Side Applications

A natural advantage of using WebLogic Server JMS is that the JMS server and the server-side applications that call into JMS can run on the *same* JVM. This, of course, removes network and serialization overhead, and consequently may greatly improve performance – especially for applications that have high message rates or large messages. Consequently, consider targeting message applications at the *same* WebLogic Server that hosts their JMS server.

Even if it is possible, co-locating JMS with applications is not always desirable. JMS sometimes is run on dedicated higher end hardware, or may be isolated to simplify integration with multiple remote applications. Also, co-locating may increase memory, disk, or CPU needs to the point where applications and JMS are in contention.

4.5.4 Co-locating Transaction Resources, JDBC Connection Pools

If an application combines more than one resource in a single transaction, the transaction becomes two-phase. If these resources can be located on the same instance of WebLogic Server, then the network overhead associated with driving the transaction is reduced. For example, imagine an EJB (EJB A) that starts a transaction, calls another EJB (EJB B), and enqueues a message. Given this arrangement, it is advantageous if EJB B and the target destination reside on the same instance of WebLogic Server.

Furthermore, if an application's transactions typically involve multiple destinations, there is an advantage to ensuring that the destinations are located on not only the same WebLogic server, but also the same JMS server. Putting the destinations on one JMS server ensures that the destinations share the same XA resource manager. As a result, the transaction monitor can optimize the transaction to handle all the operations as a single group when the transaction is committed.

4.5.5 Splitting Remote Clients Across Multiple JVMs

It is helpful to tune the number of clients that share a remote JVM. In practice, the optimal configuration for WebLogic JMS non-persistent messaging is 8 active client threads per JVM, where the client thread pool size is set to 16 (for information on tuning the client thread pool size, see section 4.1.2). For example, to run 128 remote clients, consider distributing them across 16 JVMs with 8 threads each. Note that the more JVM processes a server hosts, the more server memory required because of the additional process heaps.

Tip: Benchmarks often run multiple clients on one or more JVMs, where all JVMs run on the same server. This often does not model production applications whose clients actually run on multiple remote servers. In order to help avoid such benchmark clients from unintentionally becoming a bottleneck, it is important to tune these clients appropriately.

4.6 WebLogic JMS Clustering

4.6.1 WebLogic JMS Clustering Features

WebLogic JMS provides several clustering related features. It is important to understand them when performance is a consideration.

4.6.1.1 Distributed Destinations

Distributed destinations are a high availability, high scalability feature built into WebLogic JMS 7.0 and later. Distributed destinations allow what appears to be a single destination to a client to actually be distributed across multiple servers within a cluster.

For more information, see "Configuring Distributed Destinations" in *BEA 7.0 WebLogic Server Administration Guide*, "JMS Distributed Destination Tasks" in *BEA 8.1 Administration Console Online Help*, "Using Distributed Destinations" in *Programming WebLogic JMS*, and "Approximating Distributed Destinations" in section 7.

4.6.1.2 Server Migration

WebLogic JMS 7.0 and later provide facilities for the orderly migration of a JMS server from one server in a cluster to another via console commands or via programmatic calls to JMX MBeans. Note that this feature can be used in conjunction with an "HA Framework" (such as Veritas) and replicated disks to implement high availability automatic fail over applications.

For more information in 7.0, see the "Configuring JMS Migratable Targets" and the "Migrating JMS Data to a New Server" sections in the "Managing JMS" chapter in *BEA WebLogic Server Administration Guide*. For 8.1, look for these same sections in the "Managing WebLogic JMS" chapter of *Programming WebLogic JMS.*

4.6.1.3 Location Transparency

JMS client resource objects, including destinations and connection factories, are location transparent within the cluster. When an application establishes a JNDI naming context with any server in the cluster, it has no need to know exactly where a JMS resource exists. Instead, it just looks up its canonical name in JNDI. As a result, although a physical destination may only exist on a single server within the cluster, JMS clients attached to any server in the cluster need only use JNDI to look it up. A client can then access the destination without knowing which server hosts it.

Tip: WebLogic JMS 8.1 also provides location transparency for remote WebLogic destinations and even non-WebLogic JMS destinations. Such JMS destinations can be administratively registered in the local WebLogic JNDI. For more information, see "Simple Access to Remote or Foreign JMS Providers" in *Administration Console Online Help*.

4.6.1.4 Connection Concentrators (Connection Routing)

This feature allows a cluster to scale to support many thousands of remote JMS clients, as the work of maintaining all of the sockets can be divided among multiple servers. Each remote JMS client routes all its operations through a single connection to a single server instance within the cluster. This server is chosen according to the location(s) of the connection factory that the client looks up. See also *Connection Load Balancing* (4.6.1.5), and *Tuning Connection Factory Routing* (4.6.2).

JMS connection factory primer: A JMS client establishes its JMS connection into the cluster by looking up a particular `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory` and calling the method "`createQueueConnection()`" or "`createTopicConnection()`", respectively. Default factories are supplied, and additional customizable user-configured factories may be created as well. The default factories can be disabled on a per server basis. To disable the default factories on a particular server using the administrator console, uncheck the check-box Servers -> <server name> -> Services -> JMS -> "Enable Default JMS Connection Factories". User-configured factories can be targeted on any combination of servers in a WebLogic domain.

Note: WebLogic Server clients always use a single, shared socket between a client and a particular server, no matter how many contexts and JMS connections exist between them. This improves performance and scalability.

4.6.1.5 Connection Load Balancing

WebLogic JMS "round-robin" load balances client connections across the servers for a particular connection factory. Applications obtain access to JMS by looking up a connection factory in JNDI and invoking a "create connection" method on the factory to get a JMS connection object. The "create connection" call returns a connection that will route all JMS operations based on that connection through a fixed WebLogic Server instance. (This process is described in more detail in *Connection Concentrators*, section 4.6.1.4, above.) The WebLogic Server instance that the connection is routed through is chosen in a load-balanced manner from a list of all running servers that the connection factory is targeted to. This "round-robin" load balancing occurs each time the connection factory is looked up, and each time "create connection" is called. For example, if a connection factory, "CF1", is configured with two targets in a cluster, "server1" and "server2", and remote JMS clients (e.g. non-server-side clients) obtain their JMS connection using "CF1", then the first client routes JMS requests through "server1", the second through "server2", the third through "server1", and so on…

Server-side applications that use the application's host server JNDI context (e.g. applications that don't specify a URL when establishing their context), and which are hosted on the same WebLogic Server instance as the desired connection factory, *do not cause the connection factory to load balance* (i.e., round robin). MDBs generally fall into this category. Instead, such applications get a connection which routes JMS requests through the local server. Routing through the local server is desirable, as it avoids extra network traffic.

4.6.1.6 Connection Fail Over

After a client has looked up its connection factory in JNDI, the connection factory will continue to work, even after one or more of the client's host servers fail. After the client detects such a failure, the client requests a new connection from the connection factory, and the connection factory in turn checks an internally stored list of alternative running servers to find one that it can use. JMS clients can easily detect JMS connection failures by registering an exception listener on their JMS connection.

4.6.2 Tuning Connection Factory Routing and Load-Balancing

When a WebLogic cluster has multiple WebLogic servers, WebLogic JMS connection factory locations are used to determine WebLogic JMS connection load-balancing, routing, and fail-over behavior. This behavior may differ depending on whether a JMS application is running on a server in the same WebLogic cluster that hosts the application's JMS servers, or remotely. The following sections provide basic guidelines about where to target connection factories, and about configuring a clustered connection factory's load balancing algorithm.

4.6.2.1 Locating Connection Factories for Server Applications

Connection factories used by server side applications should usually target to the entire cluster.

For a server-side JMS application, ensure that its user configured connection factory is targeted at every server that hosts the application, or, simply target the connection factory at the entire cluster, rather than individual servers. This ensures that server-side JMS applications do not unnecessarily route their JMS requests through a third server, but instead route JMS requests through the local server.

Alternatively, target the connection factory only at the host server(s) for the JMS destination. This may prevent unnecessary routing but has drawbacks. *First*, if the JMS server is moved to another server within the cluster, then the connection factory also needs to be moved. *Second*, connection routing work is then concentrated on one server, rather than divided among the servers running the applications. *Finally*, for a distributed destination, the connection factory would need to target all of the distributed destination's host servers, which can introduce the possibility of undesirable extra-hop routing. The connection can route from the client host, through the connection host, and finally to a particular distributed destination physical instance on yet another host. For distributed destination producers, the possibility of this extra-hop is removed by enabling the connection factory "server affinity" option. The "server affinity" option disables distributed destination load-balancing in favor of using the connection's host server's physical destination instance if one exists. In effect, the "server-affinity" option moves the producer's load-balancing decision to the connection factory (see sections 4.6.1.5 and 4.6.2.3). For distributed destination consumers, the possibility of this extra-hop is removed by following the standard practice of locating such consumers on the same servers as the distributed destination.

Similarly, if default connection factories are used instead of the user-configured connection factories, ensure that the default connection factories are either enabled on all servers or only on the server that hosts the JMS server.

Tip: Unlike applications external to the cluster, applications that run on as server within a cluster need not, and should not specify a URL when obtaining the cluster's JNDI context.

4.6.2.2 Locating Connection Factories for Remote Applications

In most cases, applications remote to the cluster should use connection factories that are only targeted at the JMS destination server.

This eliminates an unnecessary additional network hop, as otherwise, client JMS requests will route through a connection factory host before reaching their destination.

If the destination is a distributed destination, the connection factory should be targeted at each server that hosts the distributed destination.

Caveat: For distributed destinations there is the possibility of undesired extra-hop routing. Consider enabling the connection factory "server affinity option" for producers, and consider retargeting consumers to run on the same servers as the distributed destination rather than remotely.

The connection can route from the client host, take an extra-hop through the connection host, and finally reach a particular distributed destination physical instance on yet another host. For distributed destination producers, the possibility of an extra-hop is removed by enabling the connection factory "server affinity" option. The "server affinity" option disables distributed destination load-balancing in favor of using the connection's host server's physical destination instance if one exists. In effect, the "server-affinity" option moves the producer's load-balancing decision to the connection factory (see sections 4.6.1.5 and 4.6.2.3). For distributed destination consumers, the possibility of an extra-hop is removed by targeting the consumers to the distributed destination's host servers, rather than running the consumers remotely.

For applications with many hundreds, or thousands of remote JVMs, target the connection factory at multiple servers in order to use them as connection concentrators.

This purposely introduces an additional network hop, so that no one server is forced to handle a very large number of sockets.

4.6.2.3 Connection Factory Load Balancing Algorithm

Weblogic Server 8.1 and later supports alternatives to the default "round-robin" load-balancing behavior for JMS connection factories, as well as for other RMI resources. Load balancing behavior is configurable on the console at *Clusters -> <your-cluster-name> -> General Tab -> "Default Load Algorithm"*. Note that this setting affects all clustered remote resources (such as stateless session beans), not just JMS connections.

The load algorithm of particular interest to JMS is "round-robin-affinity". This option may benefit remote JMS applications that use a connection factory targeted at multiple servers in a cluster, rather than targeted at just the server that hosts the desired destination. This option tells the load balancer to always choose servers for which the client *already* has a JNDI connection, if such servers host the requested resource, leading to the following benefits.

- It leaves load-balancing decisions up to the network. JMS connection routing will depend on which server the client first connects to, rather than a server chosen based on the connection factory's WebLogic server hosts.

- It prevents triple network hops that may occur due to applet security. Applet security forces applets to route all network calls through their designated host. This means that an applet JMS operation may end up in three network hops: routing through the applet's host server, then through its JMS connection server, and finally to the desired destination.

- It reduces the number of sockets opened up between client and cluster, as the client favors using servers it has already connected to for JNDI purposes. This is because when the load balancer chooses which server to host a client's JMS connection, affinity causes the load balancer to favor the server for which the client *already* has a JNDI connection.

For more information, see "Load Balancing In a Cluster" in *Using WebLogic Server Clusters*.

4.7 Using Message Paging

Paging is not enabled by default, so even persistent messages will consume server memory. Paging has a negative impact on performance, but is often unavoidable.

WebLogic Server 6.1SP2 and later versions provide the ability to page both persistent and non-persistent messages out of the server memory. Paging expands the amount of message data a WebLogic Server instance can contain without requiring an increase in the JVM heap size. If paging is enabled, messages are paged once the number of bytes and/or number of messages exceeds a configurable threshold. Paging has no effect on applications, except that it may slow performance down considerably. Paged out persistent messages may impact performance as such messages must be paged back in when a consumer requests them, causing additional I/O. Paging non-persistent messages may create less of a slow-down than you expect; in many such cases, the decline in performance may be no more than 20%.

Tip: *When configuring paging for non-persistent messages it is best to use a file store rather than a JDBC store.* There is little point in using a JDBC store, as this will perform quite poorly in comparison, without any real benefit. Non-persistent page stores do not contain data that is reused after a JMS server is shutdown, so there is no need for the high availability a JDBC store gives. The only time to use a JDBC store for this purpose is when the local file system is incapable of storing the amount of data needed.

Tip: *Message bodies may be paged out; messages headers are never paged out.* For applications that page hundreds of thousands of messages or more, it can be helpful to reduce JVM memory usage by reducing the amount of application data stored in JMS message header or message property fields.

For information about configuring paging in WebLogic JMS 8.1, see "Paging Out Messages To Free Up Memory" in *Administration Console Online Help*. In 7.0, see "Using Message Paging" in the "Managing JMS" chapter of *BEA WebLogic Server Administration Guide* (correspondingly in 6.1, use the link "Messaging Paging").

Summary:

- Persistent messages are paged into the JMS server's store.
- Non-persistent messages are paged into the JMS server's *paging* store.
- If paging is not enabled, *neither* non-persistent *nor* persistent messages will page.
- The JMS server's paging store should always be a *file* store.
- If no paging store is configured, non-persistent messages will not page.
- Paging is *not* enabled by default.
- Message bodies may be paged out; messages headers are never paged out.

4.8 Tuning the Asynchronous Pipeline (Message Backlog)

If an application uses asynchronous consumers (such as MDBs), consider increasing the WebLogic JMS Connection Factory's configured MessagesMaximum value.

WebLogic JMS *pipelines* messages that are delivered to asynchronous consumers, otherwise known as *message listeners*. This action aids performance as messages are aggregated when they are internally pushed from the server to the client. The messages backlog (the size of the pipeline) between the JMS server and the client is tunable. You can tune it by configuring the "MessagesMaximum" setting on the connection factory (the default value is 10).

Tuning up the connection factory's MessagesMaximum parameter may improve performance, sometimes dramatically. Especially consider tuning MessagesMaximum up if the JMS application defers acknowledges/commits. In this case, the suggested value is:

```
2 * (ack or commit interval) + 1
```

For example, if the JMS application acknowledges 50 messages at a time, consider tuning this value to 101. Tuning this value too high has drawbacks: it increases memory usage on the client, and it can also make a client "unfair" as its pipeline is filled with messages in favor of other clients that have not yet booted. For example, if MessagesMaximum is tuned to "10000000," then the first consumer client to connect may get virtually all messages that have already arrived at the destination, leaving other consumers without any messages, plus the first consumer will build up a large backlog of messages that may cause it to run out of memory.

For more information in 7.0, see "Asynchronous Message Pipeline" in *Programming WebLogic JMS*. Links (7.0, 8.1).

Restriction: If the asynchronous pipeline size is already set to 1, this may indicate that the application is enabling ordered redelivery, which in turn means that the pipeline must not be increased. Ordered redelivery is a feature new to WebLogic 8.1. For more information, see "Ordered Redelivery of Messages" in *Programming WebLogic JMS.*

4.9 Tuning Messaging Bridges

WebLogic Messaging Bridges are used to forward messages between JMS destinations. This section lists guidelines for tuning Messaging Bridges. For information on the uses for a Messaging Bridge, see section 4.5.2.

4.9.1 Synchronous vs. Asynchronous Mode

Set "Asynchronous Mode Enabled" to `false` *for the "Exactly-once" quality of service, and set it to* `true` *otherwise.*

If "Asynchronous Mode Enabled" is set to `true`*, consider tuning the message pipeline size.*

The "Asynchronous Mode Enabled" flag on the messaging bridge configuration determines whether the messaging bridge receives messages asynchronously using the JMS "MessageListener" interface, or whether the bridge receives messages using the synchronous JMS APIs. When a quality of service of "Exactly-once" mode is used, this setting has a large effect on performance. This is because in asynchronous mode, the bridge must start a new transaction for each message, and perform the two-phase commit across both JMS servers involved in the transaction. When receiving synchronously, however, the bridge processes multiple messages in each transaction, and the two-phase commit occurs less often. Since the two-phase commit is usually the most expensive part of the bridge transaction, the less often one occurs, the faster the bridge runs.

On the other hand, when the "Exactly-once" quality of service is *not* used, a WebLogic JMS source destination can transfer messages over the network more efficiently in asynchronous mode because it can combine ("pipeline") multiple messages in each network call. For information on tuning the WebLogic JMS pipeline size, see section 4.8. (Some non-WebLogic JMS implementations do not pipeline messages, or optimize the asynchronous listeners, in which case the "Asynchronous Mode Enabled" parameter may have little effect.)

Note: If the source destination is a non-WebLogic JMS provider, and "Exactly-once" then asynchronous mode is automatically disabled.

4.9.2 Batch Size and Batch Interval

Consider increasing "Batch Size" and setting "Batch Interval" for the "Exactly-once" quality of service. Explanation:

When "Asynchronous Mode Enabled" is set to `false` for the "Exactly-once" quality of service, the "Batch Size" parameter may be used to reduce the number of transaction commits. The "Batch Size" parameter determines the number of messages per transaction, and defaults to 10. Increasing the batch size will increase message throughput, but up to a point. The best batch size for a bridge instance depends on the specific JMS providers used, the hardware, operating system, and other factors.

The “Batch Interval” parameter is used to adjust the amount of time the bridge waits for each batch to fill before forwarding batched messages. The default “Batch Interval” is –1, which indicates “forever”. Users should experiment with the “Batch Interval” parameter to get the best possible performance. For example, if the queue is not very busy, the bridge may frequently stop forwarding in order to wait batches to fill, indicating the need to reduce the “Batch Interval”.

Note: Batch size is not used when “Asynchronous Mode Enabled” is set to `true`, and is not used unless the quality of service is “Exactly-once”.

4.9.3 Quality of Service

An “Exactly-once” quality of service may perform significantly better or worse than “At-most-once” and “Duplicate-okay”.

As described above, when the "Exactly-once" quality of service is used, the bridge must undergo a two-phase commit with both JMS servers in order to ensure the transaction semantics, and this operation can be very expensive. On the other hand, unlike the other qualities of service, the bridge can batch multiple operations together using “Exactly-once” service (see 4.9.2),

Users should experiment with this parameter to get the best possible performance. For example, if the queue is not very busy, or if non-persistent messages are used, “Exactly-once” batching may be of little benefit. On the other hand, one test yielded a 100% throughput gain when switching to “Exactly-once” with a batch size of 50.

4.9.4 Multiple Bridge Instances

If message ordering is not required, consider deploying multiple bridges.

Multiple instances of the bridge may be deployed using the same destinations. When this is done, each instance of the bridge runs in parallel, and message throughput may improve. Of course, when multiple bridge instances are used, then messages will not be forwarded in the same order they had in the source destination, so the receiving application must be prepared to handle this.

Whether the use of multiple bridges will actually improve throughput depends on several factors. Some JMS products don't seem to benefit much from using multiple bridges, although WebLogic JMS generally will significantly – especially if the messages are persistent. Also, if any resource on the server, such as the CPU or a disk, is already saturated (i.e. is busy 90-100 percent of the time), then increasing the number of bridge instances may actually decrease throughput, so be prepared to experiment in this area.

4.9.5 Bridge Thread Pool

If more than five synchronous bridges are targeted at the same WebLogic server, increase the size of the bridge thread pool to match the number of bridges instances.

To avoid competing with the default "execute thread pool" in WLS, WebLogic Messaging Bridges share a separate thread pool. This thread pool is used only in synchronous mode (eg when "Asynchronous Mode Enabled" is set to `false`). In asynchronous mode the bridge actually runs in a thread created by the JMS provider for the source destination.

For best performance in synchronous mode, there should be enough threads in this pool so that there is one available for every bridge instance that uses synchronous mode. By default, there are five threads in this pool. If more than five bridge instances are run on the same server in synchronous mode, then increasing the size of this thread pool may improve performance.

The bridge thread pool size is configured on a per server basis, on the WebLogic console see "Servers -> <server-name> -> Services Tab -> Bridge Tab -> Messaging Bridge Thread Pool Size". Alternatively, the bridge thread pool can be managed by creating a custom thread pool named "MessagingBridge". The advantage of custom thread pools is that they are easily monitored via the WebLogic console. For more information on thread pools see sections 4.1.1 and 4.1.3.

4.9.6 Durable Subscriptions

For topic source destinations, set "Durability Enabled" only when necessary.

If the bridge is listening on a Topic and it is acceptable that messages are lost when the bridge is not booted, the "Durability Enabled" flag should be disabled to ensure undelivered messages don't accumulate in the source server's store. Disabling the flag will also have the effect of making the messages non-persistent.

4.9.7 Co-locate Bridges With Their Source or Target Destination

If a messaging bridge's source or target is a WebLogic destination, deploy the bridge to the same WebLogic server as the destination.

Targeting a messaging bridge with one of its destinations eliminates the associated network and serialization overhead. Such overhead can be significant in high-throughput applications, particularly if the messages are non-persistent.

4.10 Monitoring JMS Statistics

WebLogic JMS provides a comprehensive monitoring API, based on the standard J2EE JMX interface. Statistics can be viewed through the WebLogic administrator console, which in turn uses this API. Applications may also call this API directly. BEA's developer web site, dev2dev, provides some useful JMX examples in the "Code Library/Code Samples" section. For example, it provides a simple program called "JMSStats.java" that prints virtually all of the currently available JMS statistics in table form and another program called "wlshell" that is a comprehensive command-line configuration and monitoring tool.

4.11 Additional Administrative Settings

The following section on application design contains additional information on administrative settings. Pay special attention to:

- Sorted destinations (section 5.13).
- Poison message handling (section 5.14)
- Expired message handling (section 5.15)
- Server and destination quotas (section 5.18.1).
- Sender throttling (section 5.18)

5 Application Design

This section outlines design choices that impact performance.

5.1 Message Design

5.1.1 Choosing Message Types

When deciding how to store application data in a message, consider the following costs:

- *Serializing application objects:* The CPU cost of serializing Java objects can be significant. This expense, in turn, affects JMS Object messages. You can offset this cost, to some extent, by having application objects implement java.io.Externalizable, but there still will be significant overhead in marshalling the class descriptor. To avoid the cost of having to write the class descriptors of additional objects embedded in an Object message, have these objects implement Externalizable, and call readExternal and writeExternal on them directly. (For example, call obj.writeExternal(stream) rather than stream.writeObject(obj)). Bytes and Stream messages are often preferable.

- *Serializing strings.* Serializing Java strings is more expensive than serializing other Java primitive types. Strings are also memory intensive, they consume two bytes of memory per Character, and cannot compactly represent binary data (integers, for example). In addition, the introduction of string-based messages often implies an expensive parse step in the application in order to process the String into something the application can make direct use of. Bytes, Stream, Map and even Object messages are therefore sometimes preferable to Text and XML messages. Similarly, it is preferable to avoid the use of strings in message properties, especially if they are large.

- *Message properties.* WebLogic JMS message properties are not paged when a message is paged out, thereby continuing to consume memory. They also incur an added serialization cost, both on the client and on the server. For more information on message paging, refer to section 4.7.

- *Server-side serialization.* WebLogic JMS servers do not incur the cost of serializing Object, Map, Stream, and Bytes messages. Serialization of these message types occurs on the client, but they are treated as simple byte buffers on the server.

- *Selection is sometimes expensive.* This consideration is important when you are deciding where in the message to store application data that is accessed via JMS selectors. For details, see the discussion about choosing selector fields in section 5.9.2.

5.1.2 Compressing Large Messages

Compressing large messages in a JMS application can improve performance. This reduces the amount of time required to transfer messages across the network, reduces the amount of memory used by the JMS server, and, if the messages are persistent, reduces the size of persistent writes. Text and XML messages can often be compressed significantly. Of course, compression is achieved at the expense of an increase in the CPU usage of the client.

Keep in mind that the benefits of compression become questionable for "smaller" messages. If a message is less than a few KB in size, compression can actually increase its size. The JDK provides built-in compression libraries. For details, see the "java.util.zip" package.

5.1.3 Message Properties and Message Header Fields

Instead of user-defined message properties, consider using standard JMS message header fields or the message body for message data. Message properties incur an extra cost in serialization, and are more expensive to access than standard JMS message header fields. For more information on choosing where to place message data, see "Selectors" in section 5.9.

Also, avoid embedding large amounts of data in the properties field or the header fields; only message bodies are paged out when paging is enabled. Consequently, if user-defined message properties are defined in an application, avoid the use of large string properties. For more information on message paging, see section 4.7.

5.2 Topics vs. Queues

Surprisingly, when you are starting to design your application, it is not always immediately obvious whether it would be better to use a Topic or Queue. In general, you should choose a Topic only if one of the following conditions applies:

- The same message must be replicated to multiple consumers.
- A message should be dropped if there are no active consumers that would select it.
- There are many subscribers, each with a unique selector. This is a special case (see section 5.9.6).

If message selectors are used in an application, additional factors should be considered. For information about how selectors can affect this decision, see section 5.9.4.

It is interesting to note that a topic with a single durable subscriber is semantically similar to a queue. The differences are as follows:

- If you change a topic selector for a durable subscriber, all previous messages in the subscription are deleted, while if you change a queue selector for consumer, no messages in the queue are deleted.
- A queue may have multiple consumers, and will distribute its messages in a round-robin fashion, whereas a topic subscriber is limited to only one consumer.

5.3 Asynchronous vs. Synchronous Consumers

In general, asynchronous (onMessage) consumers perform and scale better than synchronous consumers:

- *Asynchronous consumers create less network traffic.* Messages are pushed unidirectionally, and are pipelined to the message listener. Pipelining supports the aggregation of multiple messages into a single network call. For information about tuning pipelining, see section 4.8.

- *Asynchronous consumers use fewer threads.* An asynchronous consumer does not use a thread while it is inactive. A synchronous consumer consumes a thread for the duration of its receive call. As a result, a thread can remain idle for long periods, especially if the call specifies a blocking timeout.

Tip: For application code that runs on a server, it is almost always best to use asynchronous consumers, typically via MDBs. The use of asynchronous consumers prevents the application code from doing a blocking operation on the server. A blocking operation, in turn, idles a server-side thread; it can even cause deadlocks. Deadlocks occur when blocking operations consume all threads. When no threads remain to handle the operations required to unblock the blocking operation itself, that operation never stops blocking.

5.4 Persistent vs. Non-Persistent

When designing an application, make sure you specify that messages will be sent in non-persistent mode unless a persistent QOS is required. We recommend non-persistent mode because unless synchronous writes are disabled (see section 4.4.3.1), a persistent QOS almost certainly causes a significant degradation in performance.

Tip: Take special care to avoid persisting messages unintentionally. Occasionally an application sends persistent messages even though the designer intended the messages to be sent in non-persistent mode.

If your messages are truly non-persistent, none should end up in a regular JMS store. To make sure that none of your messages are unintentionally persistent, check whether the JMS store size grows when unconsumed messages are accumulating on the JMS server. Here is how message persistence is determined, in order of precedence:

- Producer's connection's connection factory configuration:
  - "PERSISTENT" (default)
  - "NON_PERSISTENT"

- JMS Producer API override on QueueSender and TopicPublisher:
  - setDeliveryMode(DeliveryMode.PERSISTENT)
  - setDeliveryMode(DeliveryMode.NON_PERSISTENT)
  - setDeliveryMode(DeliveryMode.DEFAULT_DELIVERY_MODE) (default)

- JMS Producer API per message override on QueueSender and TopicPublisher:
  - for queues, optional deliveryMode parameter on send()
  - for topics, optional deliveryMode parameter on publish()

- Override on destination configuration:
  - "Persistent"
  - "Non-Persistent"
  - "No-Delivery" (default, implies no override)

- Override on JMS server configuration:
  - No store configured implies Non-Persistent. (default)
  - Store configured implies no-override.

- Non-durable subscribers only:
  - In WebLogic 8.1 and 7.0, if there are no subscribers, or there are only non-durable subscribers for a topic, the messages will be downgraded to non-persistent. (Because non-durable subscribers exist only for the life of the JMS server, there is no reason to persist the message.)

- Temporary destinations:
  - Because temporary destinations exist only for the lifetime of their host JMS server, there is no reason to persist their messages. WebLogic JMS automatically forces all messages in a temporary destination to non-persistent.

Tip: Durable subscribers require a persistent store to be configured on their JMS server, even if they receive only non-persistent messages. A durable subscription is persisted to ensure that it continues through a server restart, as required by the JMS specification. WebLogic JMS will throw a JMSException if an attempt is made to create a durable subscription on a JMS server with no store configured.

5.5 Deferring Acknowledges and Commits

Because sends are generally faster than receives, consider reducing the overhead associated with receives by deferring acknowledgment of messages until several messages have been received and can be acknowledged collectively. (If you are using transactions substitute the word "commit" for "acknowledge.")

Deferment of acknowledgements is not likely to improve performance for non-durable subscriptions, however, because of internal optimizations already in place. (For details, see section 5.6).

It may not be possible to implement deferred acknowledgements for asynchronous listeners. If an asynchronous listener acknowledges only every 10 messages, but for some reason receives only 5, then the last few messages may not be acknowledged. One possible solution is to have the asynchronous consumer post synchronous, non-blocking receives from within its onMessage() callback to receive subsequent messages. Another possible solution is to have the listener start a timer that, when triggered, sends a message to the listener's destination in order to wake it up and complete the outstanding work that has not yet been acknowledged (provided the wake-up message can be directed solely at the correct listener).

5.6 Using AUTO_ACK for Non-Durable Subscribers

In WebLogic 8.1 and 7.0, non-durable, non-transactional topic subscribers are optimized to store local copies of the message on the client side, thus reducing network overhead when acknowledgements are being issued. This optimization yields a 10-20% performance improvement, where the improvement is more evident under higher subscriber loads.

One side effect of this optimization, particularly for high numbers of concurrent topic subscribers, is the overhead of client-side garbage collection, which can degrade performance for message subscriptions. To prevent such degradation, we recommended allocating a larger heap size on the subscriber client. (For instructions on tuning JVM memory, see section 4.3.) For example, in a test of 100 concurrent subscribers running in 10 JVMs, it was found that giving clients an initial and maximum heap size of 64MB for each JVM was sufficient.

5.7 Alternative Qualities of Service, Multicast and No-Acknowledge

WebLogic JMS 6.0 and above provide alternative qualities of service (QOS) extensions that can aid performance.

- *MULTICAST_NO_ACKNOWLEDGE*. Non-durable topic subscribers can subscribe to messages using the MULTICAST_NO_ACKNOWLEDGE acknowledge mode. If a topic has such subscribers, the JMS server will broadcast messages to them using multicast mode. Multicast improves performance considerably and provides linear scalability, as the network only needs to handle only one message, regardless of the number of subscribers, rather than one message per subscriber. Multicast messages may be lost if the network is congested, or if the client falls behind in processing them. Calls to "recover()" or "acknowledge()" have no effect on multicast messages.

  Note: On the client side, each multicasting session requires a dedicated thread to retrieve messages off the multicast socket. Therefore, you should increase the JMS client-side thread pool size to adjust for this. For information on tuning the client-side thread pool size see section 4.1.2.

  Tip: This QOS *extension* has the same level of guarantee as some JMS implementations *default* QOS from vendors other than BEA WebLogic Server for non-durable topic subscriptions. The JMS 1.0.2 specification specifically allows non-durable topic messages to be dropped (deleted) if the subscriber is not ready for them. WebLogic JMS actually has a higher QOS for non-durable topic subscriptions by default than the JMS 1.0.2 specification requires. For more information about non-durable subscribers, see section 5.6.

- *NO_ACKNOWLEDGE*. A no-acknowledge delivery mode implies that the server gives messages to consumers, but does not expect acknowledge to be called. Instead, the server pre-acknowledges the message. In this acknowledge mode, calls to recover will not work, as the message is already acknowledged. This mode saves the overhead of an additional network call to acknowledge, at the expense of possibly losing a message when a server failure, a network failure, or a client failure occurs. Note that if an asynchronous client calls "close()" in this scenario, all messages in the asynchronous pipeline are lost.

  Tip: The tip about **"**MULTICAST_NO_ACKNOWLEDGE," above, also applies here**.**

  Tip: Asynchronous consumers that use a NO_ACKNOWLEDGE QOS may wish to tune down their message pipeline size in order to reduce the number of lost messages in the event of a crash. For information on turning messaging pipeline size, see section 4.8.

For more information, see "WebLogic JMS Fundamentals" in *Programming WebLogic JMS*. Links: 6.0, 6.1, 7.0, and 8.1. See also "Using Multicasting" in *Developing a WebLogic JMS Application.* Links: 6.1, 7.0, and 8.1. See also "JMS Topic --> Configuration --> Multicast" in the Administration Console Online Help.

5.8 Transacted vs. User Transaction Aware Sessions

A transaction is used to guarantee the atomicity of a set of operations. Atomicity means that operations within a transaction are guaranteed to either all succeed, or all roll back as if they never occurred. If a transaction involves only a single operation (such as a send), there is little point in using a transaction, as a single operation is already atomic and the transaction itself incurs extra overhead. *Transactions should be used only when an application requires atomicity across multiple operations.*

A user transaction is a transaction started by an EJB container, by a connector container, or explicitly by the application. For the purposes of this white-paper, global transactions, JTA transactions, and container managed transactions (CMT) are all considered to be user transactions.

JMS supports transactions via either "transacted" sessions or "user transaction aware" sessions. Setting aside performance considerations, it is important to consider the differences between the two. There are performance advantages in certain cases when choosing transacted sessions, but the choice is usually limited by other considerations.

A JMS session is "transacted" if the first parameter of either

```
QueueConnection.createQueueSession()
TopicConnection.createTopicSession()
```

is `true`. Transacted sessions ignore user transactions in favor of their own internal transaction, sometimes referred to as a "local" transaction. A transacted session's inner transaction is limited in scope to the send and receive operations on the session. A transacted session therefore *cannot* participate in a user transaction, such as an EJB or MDB container managed transaction, or a transaction explicitly started by an application.

> **Tip:** Unlike most JMS vendors, WebLogic JMS transparently supports a true transactional guarantee on a transacted sessions even if the sends and/or receives within the same local transaction span multiple servers (referred to as two phase, or distributed transactions).

A JMS session is considered "user transaction aware" if it is **not** transacted and its connection factory is either an XAConnectionFactory or has configured its "users transactions enabled" setting to `true`. The default connection factories "javax.jms.QueueConnectionFactory" and "javax.jms.TopicConnectionFactory" set "user transactions enabled" to be true, while the default connection factory "weblogic.jms.ConnectionFactory" sets this value to false. Unlike transacted sessions, a user transaction aware session participates in the thread's current transaction. *Therefore, unlike "transacted" sessions, a "user transaction aware" session's send and receive operations may participate in a JTA transaction with one or more additional transaction aware operations.* These additional operations are commonly sends/receives on another JMS session, EJB invokes, or database (JDBC) calls.

A WebLogic MDB can be user transaction aware, but not transacted. Conversely, a WebLogic ServerSessionPool can be transacted, but not user transaction aware. For more information on MDBs vs. SSPs see section 5.11.

Transacted sessions are generally better performing than user transaction aware sessions if the client session is not running directly on the same WebLogic Server instance as one of the involved destinations. This is because a transacted session transaction begins and ends at the server that hosts the session's connection, rather than the client, reducing network overhead for remote clients. If the session is running directly on the same server as one of the involved destinations, there is no performance difference between the two approaches.

For more information on transactions or JMS refer to "Using Transactions with WebLogic JMS" in *Programming WebLogic JMS* (links 6.0, 6.1, 7.0, 8.1), to the transactions section of the JMS FAQ (available at BEA's developer web site), and to *Programming WebLogic JTA* (links 6.1, 7.0, 8.1).

5.9 JMS Selectors

When it comes to high performing applications, the use of selection must be carefully considered. Often, the answer is to avoid the use of selectors. This section addresses the following:

- What are JMS selectors?
- Which is better: multiple destinations or a single destination with a selector?
- Should an application that uses selectors use topics or queues?
- How can I make a selector more efficient?
- Which fields are best for selectors to use?
- Replacing selectors with temporary queues.

5.9.1 Selector Primer

JMS selectors are optional message filters for JMS consumers. Their SQL-like syntax can be used to form complex expressions based on JMS message fields and/or JMS message properties. In addition, WebLogic JMS provides an extension to the standard selector syntax that allows inclusion of xpath expressions on XML message bodies.

A javax.jms.QueueConsumer or javax.jms.TopicSubscriber may specify a selector when they are first created; an MDB may specify a selector inside its XML descriptor. As per the JMS specification, selectors have special behavior with respect to durable subscribers: changing the selector on a durable subscription causes all messages accumulated under the old selector to be deleted – essentially restarting the subscription. In all cases, the selector is static; it cannot be changed without closing the current consumer and creating a new one. This means that selectors on MDBs can't be changed except through undeploying the MDB, and then redeploying it with a changed descriptor.

In WebLogic JMS, all filtering occurs on the server side with one exception: multicast topic subscribers use client side filtering. *If your application is using multicast topic subscriptions, the selection cost often is not an issue, as it is paid once per message per client on the client's CPU and the server incurs no selection cost.* Multicast topic subscriptions are a WebLogic JMS extension. For more information on multicast subscribers, see section 5.7.

Selectors are CPU intensive, and certain applications can cause selector evaluation to consume the majority of a server's CPU. For queues, every synchronous receive request forces a scan of each message in the queue until a matching message is found. Also for queues, when a message arrives on the queue it must be evaluated against the selectors of the current blocking synchronous receivers and of the started asynchronous listeners until one matches. For topics, selection evaluation occurs when the message is published – once for each subscriber.

5.9.2 Choosing Selector Fields

WebLogic JMS selector expressions may reference three types of fields:

- Fields in the message header
- Fields in the message properties
- Fields in an XML message body

5.9.2.1 Selecting by Message Header Fields

Message header fields are the least expensive fields that can be accessed by selectors. We recommend designing a selector that only accesses these fields. The message header fields include:

- Set by JMS provider on produce:
  - JMSMessageID *
  - JMSTimestamp *
- Set by JMS provider on redelivery:
  - JMSRedelivered *
- Set by JMS application:
  - JMSCorrelationID
  - JMSType
  - JMSPriority *
  - JMSExpiration *
  - JMSDeliveryTime (WebLogic 6.1 extension) *

Changing the fields marked with an asterisk (*) directly using the javax.jms.Message interface has no effect. These fields are set internally by JMS when the message is first produced, as per the JMS specification.

Tip: If a selector expression involves both message header fields and non-message header fields, you should take advantage of WebLogic's left-to-right expression evaluation to minimize access to non-message header fields. For example, it is less expensive to use the expression "JMSPriority > 3 AND (expensive expression)" than it is to use the expression "(expensive expression) AND JMSPriority > 3".

5.9.2.2 Selecting by Message Property Fields

Message property fields, which generally consist of arbitrary names and values set by an application, are more expensive to access than message header fields, but less expensive to access than fields in an XML message body.

5.9.2.3 Selecting on Fields in an XML Message Body

WebLogic JMS provides a selector extension to handle XML xpath expressions. An example of a selector that contains an xpath expression is:

```
"JMS_BEA_SELECT('xpath','/Order/Manifest/Item[2]/ID/text()') = '208'"
```

Selectors with xpath expressions are the most expensive expressions to evaluate, as they necessarily involve parsing and evaluating XML. If a selector contains an "and" clause, make sure that any xpath selector is on the right side of the clause. In other words, whenever possible, take advantage of WebLogic JMS Server short-circuit expression evaluation to avoid more expensive parts of the expression.

Note: *Xpath selectors and queue message paging can conflict*. If message paging is enabled and a queue message is paged out, it must be paged back in to evaluate it against new queue selectors. This process can yield greatly increased paging activity, slowing down performance considerably. The conflict does not occur with JMS topics, as topic selectors are evaluated as soon as they are published, even before the message is paged out.

5.9.3 Using Primitives for Selector Expressions

Favor the use of primitive operators such as "=", ">", "<", over the use of the more complex operators, such as "like" and "in".

5.9.4 Topic vs. Queue Selectors

Which performs better with multiple consumers that have selectors, a queue or a topic? For topics, the number of selector evaluations per message is predictable – the number is directly proportional to the number of subscribers, unless indexed subscribers are used (see section 5.9.6), in which case there is one selector evaluation per message. For queues, the number of evaluations per message is not predictable. The number may be as low as one, if every consumer selector matches every message, or the number may be a relatively large number depending on how restrictive the selectors are and how many messages are in the queue.

Queue selector performance depends on whether or not receivers are "fast" and equal in speed, and if the messages tend to match evenly to the selectors. If receivers are fast and the selectors are fair, then the queue is generally empty, and the JMS server can simply hand the message to the first waiting receiver that matches the selector. If queue receivers are slow, or some selectors are more restrictive than others, then the receivers fall behind, and the JMS server must rescan the entire queue each time a receiver goes back to get more messages to match selectors. This incurs a potentially exponential cost. With asynchronous receivers, you can alleviate this somewhat by increasing the size of the pipeline of unprocessed messages between the server and the receiver. The default size of the pipeline is 10. Increasing the size of the pipeline allows asynchronous receivers to fall farther behind without incurring the cost of queue scans, but once the pipeline is full, then queue scans are required again. For more information about pipelines, see section 4.8.

So if your receivers are extremely well behaved, and never fall behind, then the queue solution is likely better, otherwise the topic solution may be better.

5.9.5 Avoiding Selection Completely

Two common design patterns often serve to avoid using selectors completely.

Partition messages using multiple destinations instead of using a single destination and selectors. For example, suppose an application selects messages based on color. The application could instead have a destination for each color. When a producer sends a "blue" message, it sends the message directly to the "blue" queue, and "blue" consumers need only listen to the "blue" queue.

Use temporary queues to implement request/response. A common design pattern in a messaging application is request/response. Applications sometimes implement request/response by having all requesters share a common response queue. Requesters attach a unique correlation-id to a request, responders set the same correlation-id value on the response, and then requesters select on the shared response queue based on their unique correlation-id to get their responses. To avoid the use of selectors, request/response applications can use temporary queues instead. For information about using temporary queues for request/response, see section 5.18.6.

Use indexed topic subscribers: See section 5.9.6.

5.9.6 Indexed Topic Subscribers

For a certain class of applications, WebLogic JMS can significantly optimize topic subscriber selectors by indexing them. These applications have messages that contain one or more unique identifiers and thousands of subscribers that filter based on these identifiers. A typical example is an instant messaging application where each subscriber corresponds to a different user, and each message contains a list of one or more target users. This optimization may yield as much as 7x gain in throughput and is activated by using a specifically formatted subscriber selector.

This optimization is available in WebLogic 8.1, 7.0SP3, 6.1SP5. For information see "Indexing Topic Subscriber Message Selectors To Optimize Performance" in *Programming WebLogic JMS*.

5.10 Looking up Destinations

The action of looking up a destination is expensive. One way to minimize lookups of a destination is to use 'pinned' producers. These are producers whose target destination is set when they are created rather than passed in as a parameter to the send() call. Using pinned producers also allows the JMS server to cache its destination once, instead of sending it on each produce. Another way to minimize lookups of a destination is to cache lookup results in the application or in a server-wide pool (see section 6 for an example).

The JMS API provides vendor-specific destination finders that are generally faster than a JNDI (Context) lookup. Even so, because these methods have parameters that require vendor specific formats, and because destinations lookups tend to be infrequent in most applications, destination lookups are generally best done via JNDI. But if an application performs frequent destination lookups, it may help to consider using them:

```
QueueSession.createQueue(String dest)
TopicSession.createTopic(String dest).
```

The WebLogic JMS format for the `dest` parameter is `(jms_server_name + "/" + destination_name)`. If the Session's connection is hosted on the same server as the destination, the syntax can be simplified to `("./" + destination_name)`. This simplified syntax is available in 8.1, 7.0, 6.1SP3, or as a patch CR072612 on 6.1SP2.

Note: In WebLogic JMS 6.0 and later, the `createQueue()` and `createTopic()` methods do not create a destination if it doesn't already exist, despite what their name implies. Previous versions of WebLogic did so, but in violation of the JMS 1.0.2 specification. Instead, WebLogic JMS 6.0 and later provide alternative WebLogic specific helper methods for this purpose, see the javadoc for weblogic.jms.extensions.JMSHelper (links 6.0, 6.1, 7.0, 8.1).

5.11 MDBs vs. ServerSessionPools

Message driven bean pools (MDB deployments) and server session pools (SSPs) are sets of asynchronous message consumers. Message driven beans are enterprise Java beans (EJBs) that are activated by the arrival of a JMS message. MDBs were introduced in WebLogic 6.0, with substantial changes in 6.0 SP1 to support foreign JMS providers. SSPs are an optional facility of the JMS 1.0.2 specification similar to, but more limited than, message driven beans. Unlike MDBs, SSPs are available in all versions of WebLogic JMS. Here is direct comparison of MDBs and SSPs:

MDBs and SSPs both provide distinct advantages in a queuing application:

- They both pool (cache) their resource for re-use. Pooling eliminates the overhead of opening and closing frequently used resources.
- They both encapsulate asynchronous applications nicely; the application is only active and using server resources when a new message to be processed arrives.
- They both inherently provide parallelism to an application, as messages may be processed concurrently.

MDBs provide some additional advantages over SSPs:

- Message driven beans can be transactional. With MDBs it is possible to combine the asynchronous arrival and processing of a message within the same JTA transaction as a JDBC call or any other "XA" aware resource. This is not possible with server session pools.
- Message driven beans may be driven by JMS implementations other than WebLogic JMS (foreign providers). MDBs therefore provide a convenient method for integrating foreign resources into WebLogic applications.
- Unlike SSPs, MDBs are not limited to residing on the same server as the destination. If the remote destination is unavailable, the MDB will periodically re-attempt connecting automatically. Of course an MDB's performance improves if it is placed on the same server instance as the destination that drives it.
- MDBs are required by the J2EE 1.3 specification. Server session pools are not required; they are an optional part of the JMS 1.0.2 specification.

SSPs provide two features that MDBs do not have:

- A ServerSessionPool can have multiple ConnectionConsumers. This allows multiple consumers to drive a single MessageListener. This also allows a single MessageListener to listen on multiple destinations.
- Although they do not support JTA transactions, SSPs can use "transacted" sessions (for more on transacted sessions, see section 5.8).

Note: Topic-driven MDBs have vendor-specific semantics. In WebLogic, MDBs that listen on a topic currently receive one message per deployment [as of 6.0SP1]. If you have an MDB with multiple instances on a single server listening on a topic, it will receive only one copy of a published message regardless of the number of instances. If you have an MDB deployed across multiple machines listening on a topic, then each deployment will receive a copy of any published message on that topic. In other words, you will get multiple copies of the message distributed evenly once to each of your MDB deployments. If you need queue semantics for a topic, configure a messaging bridge to forward the topic's messages to a queue – this approach enables multiple MDB pools to share a durable subscription.

5.12 Producer and Consumer Pooling

Just as consumers are pooled using an MDB pool, producers and consumers can be pooled for application use. Producer and consumer pools are useful if many JMS resources are being looked up, created, and closed per second on a JVM. The overhead of finding, creating, and closing such resources is higher than the overhead of sending or receiving a non-persistent message. There is no standard for JMS resource pools, but WebLogic Server 8.1 provides them for server-side applications. Alternatively, application programmers can develop their own custom pools.

For server-side applications, WebLogic 8.1 provides advanced *built-in* "resource reference JMS pooling" (RRJP). This pooling is transparently activated when resource references are used to access JMS resources. In addition to WebLogic JMS resources, RRJP can pool other vendor's JMS resources. RRJP may not perform as well as a custom pool, but has the advantage of requiring no additional code, and of providing automatic transaction enlistment for foreign JMS vendor resources. You can monitor RRJP on the console: "<server-name> → "Monitoring Tab" → "JMS Tab" → "Monitor Pooled JMS Connections". For more information, see "Using JMS with EJBs and Servlets" in *Programming WebLogic JMS*.

For client-side applications, or server-side applications that cannot take advantage of WebLogic 8.1 built-in JMS pooling, you can write a custom pool. In the appendix (section 6), there is sample code for a "pinned" queue producer pool, in which the connection factory, connection, session, producer, message, and queue are all cached for fast reuse. A "pinned" producer is a producer that sets its destination upon creation, and cannot change it afterwards. This pool could be initialized on a WebLogic Server via a startup class or a load-on-start servlet. Keep in mind that static classes may be garbage collected if there are no references to them, so it is best to make sure the application server has a permanent pointer to them in some manner.

Tip: In addition to pooling producers and consumers, it is also sometimes cache JNDI lookups, such as for connection factories and destinations. See section 5.10.

Advanced notes on resource reference JMS pools (RRJP):

For best performance, use an XAConnectionFactory based pool for applications that send or receive inside a user transaction, otherwise use a non-XA ConnectionFactory. An XAConnectionFactory based RRJP will use an internal transaction if no transaction is present, which yields a significant performance penalty with no increase in transactional integrity.

Resource reference pools work best with pinned producers. Each RRJP pooled session object caches its last producer. For new producers created after obtaining a pooled session, the session checks to see if the destination on the cached producer matches, and if so, re-uses it. Therefore, the RRJP is efficient for sender applications that always send to the same destination. On the other hand, if the application sends messages to many different destinations, then the pooling is less efficient because it has to open a new producer each time. The work-around is to use non-pinned "anonymous" producers (created using null as the Destination). The anonymous producer is then cached, saving the need to create new producers, but there's additional overhead on the JMS server for anonymous/non-pinned producers (see section 5.10).

5.13 Sorted Destinations

WebLogic JMS 6.0 and above provide the ability to sort destinations based on message header field values or even message property values.

5.13.1 In Most Cases, Use FIFO or LIFO

Destinations are sorted in FIFO (first-in, first-out) order by default, which sorts ascending based on each message's unique JMSMessageID. To sort in LIFO (last-in, first-out) order, use the console to create a JMS sort key that is based on the JMSMessageID in *descending* order. Sorting in any order other than FIFO or LIFO may result in degraded performance, as the JMS server has to scan the queue to find the correct place to insert new messages rather than just placing them at the end or beginning. On the other hand, sorting can improve the performance of the particular messages that are sorted with a higher precedence.

5.13.2 Consider Sorting Using Message Header Fields

If sorting FIFO or LIFO is not convenient, then try to sort based on the message header fields, rather than message property fields. Message property fields are more expensive to access and more expensive to serialize. The message header fields are JMSMessageID, JMSTimestamp, JMSRedelivered, JMSCorrelationID, JMSType, JMSPriority, JMSExpiration, and JMSDeliveryTime. The first three fields are set by JMS itself, the latter three fields are optionally set by the application via the producer. Changing the latter three fields on the Message directly has no effect, as the JMS producer will override them (per the JMS specification). The optional JMSType and JMSCorrelationID fields are set by the application directly on the message. The JMSDeliveryTime feature was added in WebLogic 6.1.

5.13.3 Consider Sorting on Expiration Times

Applications that set message expiration times may want to sort by JMSExpiration to bias receivers towards processing messages that are due to expire soonest.

5.13.4 Consider Sorting on Delivery Times

If delivery times are used, sorting by JMSDeliveryTime is often better than sorting in LIFO or FIFO order. Delivery times are sometimes referred to as "birth times". Delivery times schedule when a message gets delivered. If messages are scheduled, this defeats the FIFO and LIFO optimizations discussed above in section 5.13.1, as scheduled messages are likely added to a destination out of their JMSMessageId order. If scheduled messages are used, the best sort option is to sort by JMSDeliveryTime, usually in ascending order. This sort order ensures that newly scheduled (born) messages are placed at the end of the queue, and that consumers, which dequeue from the front of the queue, get messages that have been in the queue the longest.

Delivery times are a feature specific to WebLogic JMS 6.1 and later, and are not a part of the JMS specification. For more information, see section 5.16.

5.13.5 Effective Sorting

Sorting is not generally effective unless there is a message backlog. Messages are sorted with respect to other messages in the same destination. If the current message count for a destination is typically one or less, then sorting is not effective as there are no other messages with which to sort. If the current message count for a destination is small, then consider avoiding sorting altogether as there is little benefit.

5.14 Poison Message Handling (Rollbacks & Recovers)

Poison messages are messages that a receiver must reject. Typically, a message is rejected due to a problem with the message itself, but a message can also be rejected due to a temporary resource outage. A poison message can slow down an application:

- Once a message is rolled back or recovered by a receiver, JMS is required to immediately redeliver the message to the next available receiver. The redelivery forces a receiver to deal with the problem message again even though the receiver is likely not ready for it. Administrators sometimes report this as an "endless loop", and may incorrectly diagnose the problem as a problem with the messaging system rather than as a problem with their application.

- Rollbacks and recovers can be relatively expensive. If they occur on an asynchronous consumer, the poison message and the subsequent messages following it in the pipeline may need to be flushed, re-inserted into their originating destination, and pipelined again.

Three WebLogic JMS specific features that were added to WebLogic Server 6.1 address poison message handling: *redelivery delays*, *redelivery limits*, and *error destinations*. If a message is rolled back or recovered, the application may specify a *redelivery delay*, which tells JMS to defer redelivering the message for some amount of time. The call to rollback or recover returns immediately, but the message stays out of circulation until the application thinks it is ready to try again. If a *redelivery limit* is set on a destination, and a message is rolled back or recovered more times than this limit, that message is removed from the queue and forwarded to its destination's *error destination*. The application is assumed to have some special code that reads the *error destination* and takes appropriate action. If the *error destination* is not configured, and a message exceeds its redelivery limit, that message is simply deleted. For more information, see "Managing Rolled Back or Recovered Messages" in *Programming WebLogic JMS* (links 6.1, 7.0, and 8.1).

Tip: Asynchronous JMS applications that have access to their Connection may pause message delivery by calling "javax.jms.Connection.stop()", and can later call "javax.jms.Connection.start()" to continue processing messages. Note that MDB and ServerSessionPool applications (see section 5.11) do not have access to their underlying connection.

Tip: Messages with a redelivery delay do not prevent other messages behind the delayed message from being delivered. Redelivery delay is very effective when rolling back a single poison message. It is less effective when rolling back a group of poison messages, or when possibly dealing with a resource availability problem in the application resulting in the rollback/recover and redelivery of all messages.

5.15 Expired Message Handling

The JMS API allows applications to set an expiration time for a message. Setting message expirations tells the messaging server not to deliver the message after a given time. The JMS specification doesn't define the handling of an expired message – most commonly automatically deleting the message. Deleting old messages, in turn, frees up server memory and eliminates the overhead of handing out-dated messages to consumers.

A JMS application's message producers can programmatically set the expiration time for each individual message using the `javax.jms.MessageProducer.setTimeToLive()`, as a parameter to `javax.jms.QueueSender.send()`, or as a parameter to `javax.jms.TopicPublisher.publish()`. Alternatively, WebLogic provides the ability to set an expiration time override for all messages on a destination (on the console, use *destination name -> Overrides tab -> Time To Live Override*), as well as a default expiration time for senders and publishers on the connection factory (on the console, see *JMS Connection Factory -> General tab -> Default Time To Live*).

WebLogic JMS 8.1 provides advanced expired message handling. Unlike previous versions, which *passively* expired messages, 8.1 *actively* expires messages by actively sweeping its destinations to check for expired messages. The sweep interval defaults to 30 seconds, and is configurable on the console at *JMS Server -> General tab -> Expiration Scan Interval*. WebLogic versions prior to 8.1 do not actively check for expired messages – such messages are cleaned up passively only when an application attempts to dequeue them or during the JMS Server boot.

In addition, WebLogic JMS 8.1 provides the ability to log each expired messages or even ***redirect expired messages to another destination***. This is configurable on the console at the JMS Server -> JMS Destinations -> Expiration Policy tab. For more information, see "Expiration Policy" in the *Administration Console Online Help* for topics, for queues, and for templates.

Tip: Weblogic JMS provides a helper class, weblogic.jms.extensions.Schedule, for applications that need to set an expiration to a specific time. This helper class understands "cron-like" recurring schedules. See javadoc links 6.1, 7.0, and 8.1.

Note: As per the JMS specification, the setJMSExpiration() method on javax.jms.Message is not intended for application use, it is reserved for internal use by JMS vendors. Instead, applications may set the expiration time of the message via the methods outlined above.

5.16 Setting Message Delivery Times

Setting the delivery time for a message sets the time that the message will be made available to a receiver. This is sometimes referred to as a message's "birth time". This feature is available via WebLogic JMS 6.1 (and later) extensions.

Delivery times are related to performance in two ways:

- Implementing delivery times external to a queuing system is not easy: the resulting application can be complex, slow, and prone to error.
- Scheduling the delivery of a message can improve performance. For example, message processing that is not time critical can be delayed until the host has spare CPU time to handle it (such as the early morning hours).

The delivery time of a message can be set via the JMS producer by casting a WebLogic JMS producer to `weblogic.jms.extension.WLProducer` and calling the `WLProducer.setTimeToDeliver()` method. The delivery time of a message may also be set administratively via a configurable override on the destination. In addition to a relative time, a delivery time can be administratively or programmatically specified via a flexible cron-like schedule. Schedule's can express such abstractions as "every hour on the hour", or "weekdays at 5:30 PM". Schedules may be configured administratively using the aforementioned destination override, or, alternatively, schedules are available programmatically to clients in the form of static helper class methods.

For more information, see "Setting Message Delivery Times" in *Programming WebLogic JMS* (links 6.1, 7.0, and 8.1). For information on programmatic schedules, see the javadoc for `weblogic.jms.extensions.Schedule` (links 6.1, 7.0, and 8.1).

Tip: If delivery times are used in an application, consider sorting destinations based on delivery times. For more information, see section 5.13.4.

5.17 Network Bandwidth Limits

It is not uncommon for designers to heavily overestimate the capability of their available network bandwidth. The network may become a significant performance bottleneck and the JMS server cannot run at full capacity.

To ensure adequate network bandwidth, consider making sure applications and/or servers do not share the same network wires. For example, ensure each server gets its own line into a high-speed switch, or ensure that web network traffic enters a server on a different network card than application traffic and server-to-server traffic.

If an application depends on very high message throughput, we recommend calculating the application's network bandwidth requirements and incorporating the results during the application's design. The following examples illustrate this process. These examples each assume a basic two-tier client/server application with the following:

- No JMS requests routing through a web server proxy or an applet's server
- No JMS operations on destinations that are *not* on the same server as the server the JMS connection is connected to
- No use of distributed destinations
- A shared 100 Mbit/sec network
- Efficient use up to 80% of the network

Example One:

- 1 MByte message size
- 1 Sender and 1 Receiver
- = 100 Mbit per sec * .8 / 8 bits per byte / 1 MByte / 2 (one sender and one receiver)
- = 5 messages per second

Example Two:

- 50 KByte message size
- 1 publisher and 9 subscribers (not multicast)
- = 100 Mbit per sec * .8 / 8 bits per byte / 50 KByte / 10 (one sender snf 9 subscribers)
- = 160 messages per second aggregate
- = 18 messages per second per subscriber

Example Three:

- 50 KByte message size
- 1 publisher and 9 subscribers (multicast)
- = 100 Mbit per sec * .8 / 8 bits per byte / 50 KByte / 2 (one sender and any number of multicast subscribers)
- = 800 messages per second aggregate
- = 89 messages per second per subscriber (any number of multicast subscribers)

Notes:

- ✓ The message sizes in the examples above are large enough such that the relatively small overhead of acknowledgment, receiver request, and producer response is ignored. As a rule of thumb, assume these take up 256 or even 512 bytes each. Although they typically take much less than this, these values make for a good approximation.

- ✓ Transaction use will generate additional network traffic, and can itself take up a significant network bandwidth at high message rates; even more so if the transactions are distributed across multiple resources on multiple servers.

- ✓ The calculations above do not take into account the limitations of the server's hosting the client and server JVMs. Often for non-persistent messages, the CPU is the bottleneck, especially if some client JVMs share the same CPU as the server's JVM. For persistent messages the database or hard-drive may be the bottleneck.

- ✓ Database-based JMS stores will generate even more network traffic for persistent messages. Remember to factor in that the message must be sent to the database, and that the database must send a response back.

5.18 Overflow Conditions and Throttling or "What to do When the JMS Server is Running out of Memory"

In most messaging systems, senders are faster than receivers because they are usually required to do less work. At a minimum, a receiver incurs additional overhead by issuing the obligatory acknowledgment call for each message. If not limited in some way, this imbalance can lead to overflow conditions that degrade response times, and may even cause a WebLogic Server instance to run out of memory.

A technique called *throttling* addresses the issue of senders outpacing receivers. Throttling also addresses the issue of senders starving the receivers out of their fair share of CPU and other server resources. When deciding whether to implement throttling, consider the following issues:

5.18.1 Always set Quotas

As of WebLogic JMS 6.0, there are configurable quotas on JMS destinations and JMS servers; namely the BytesMaximum and MessagesMaximum settings**.** When a producer send exceeds configured quotas, WebLogic JMS throws the exception "javax.jms.ResourceAllocationException", and the message is not sent. In this case, an application program's typical remedial action is to retry the send after a delay.

See also section 5.18.5, on causing senders to block until quota is available.

Tip: *It is good practice to always configure quotas, to prevent a server from running out of memory.*

Note: The "MessagesMaximum" setting on connection factories is **not** a quota but instead tunes the size of the asynchronous message pipeline. See the configuration section of this guide for details on this setting (section 4.8).

5.18.2 Message Paging

Paging reduces the amount of JVM memory required to store a message. WebLogic JMS 6.1SP2 and later support paging messages to disk once the total number of bytes or total number of messages exceeds a configurable threshold. Even non-persistent messages can be paged.

Note: Messaging performance degrades once paging takes effect.

For more information, see section 4.7.

5.18.3 Tune Flow Control

WebLogic JMS 7.0 and later provide tunable flow control to slow down overactive producers. This flow control takes effect if a configurable messages or bytes threshold is exceeded, at which point producers are forced to slow down by delaying the time when their calls to produce a message return. Flow control is configurable via a producer's connection factory.

See also *Blocking Send Timeout* in section 5.18.5.

5.18.4 Check Network Bandwidth

It is not uncommon with messaging applications to fail take into account network bandwidth bottlenecks. For a detailed discussion, see section 5.17.

5.18.5 Increasing the Blocking Send Timeout

WebLogic JMS 8.1 will temporarily block producer send operations if quota is reached. If the quota condition eases during the time the producer blocks (e.g., a message times out or a consumer consumes a message), the send will still succeed. This enhances the performance of applications that retry sends on quota failures. By default, producers block up to 10 ms. This blocking time is configurable on the console at *JMS Connection Factories -> Flow Control tab -> Send timeout*. By default, blocked senders are served on a first-come, first-served basis, but they can optionally be served on a pre-emptive basis, where the producers with smallest messages are served first. This blocking policy is configurable on the console at *JMS Server -> Thresholds & Quotas tab -> Blocking Send Policy*.

See also *Tuning Flow Control* in section 5.18.3, and *Always Set Quotas* in section 5.18.1.

Tip: JMS vendor comparison benchmarks often use quotas in combination with high blocking send timeouts to achieve flow control for steady-state benchmarking. Failure to configure this combination will often yield low performance in such benchmarks.

5.18.6 Using a Request/Response Design

Request/response is an effective and commonly used throttling method that is incorporated into an application's design. In request/response, a requester is blocked from sending again until it gets some kind of response from a receiver. A typical request/response design is:

- The requester creates a temporary queue on which to listen for responses. (See the JMS API call javax.jms.QueueSession.createTemporaryQueue()).
- The requester includes the temporary queue in the JMSReplyTo field of its JMS message.
- The requester optionally includes a correlation-id in the JMSCorrelationId field, JMSType field, JMSPriority field, or a property field of its JMS message. This allows the requester to have multiple outstanding requests that are differentiated by correlation-id.
- The requester sends its JMS message to the receiver's destination.
- The requester blocks on its temporary queue to wait for a response message from the receiver, or releases its thread and posts an asynchronous consumer on its temporary queue to wake itself up.
- The receiver operates on the requester's JMS message, gets the requester's temporary queue from the Reply field of the message, and sends a response message with an appropriate correlation-id back to the requester's temporary queue.

Tip: JMS provides a helper class that encapsulates request/response behavior. This helper class may be suited to simpler applications, see the javadoc for javax.jms.QueueRequestor and javax.jms.TopicRequestor.

Tip: Performance sensitive applications can sometimes reduce the number of response messages by generating a response per X requests rather than per each request. Often referred to as a windowing algorithm, this pattern is often implemented in a way that ensures that requesters rarely block. Reduced requester blocking is achieved by requestors marking every Xth request with a "response required" flag, by responders generating responses only for "response required" requests, and by requesters blocking only if no response is received after 2X requests.

5.18.7 Designing to Reduce Send or Acknowledge Rates

It is sometimes possible to change an application's design so that it sends fewer messages, or aggregates several messages into one. It is sometimes also possible to defer acknowledging or committing receives until several messages have been received (see section 5.5).

5.18.8 Tuning Sender and Receiver Counts

Performance can often benefit by increasing the number of receivers and/or decreasing the number of senders. Sometimes increasing the number of senders can help, as a WebLogic JMS can aggregate multiple persistent sender requests which reduces overall disk usage, effectively giving the receivers more resource to work with. Tuning sender and receiver counts can often be achieved simply by modifying the number of instances in EJB and MDB deployments, and/or by modifying thread pool sizes (see sections 4.1.3 and 4.1.4).

5.18.9 Partitioning and/or Clustering

Many times, an application can be scaled by partitioning or clustering. For more information, see section 4.5, "Partitioning Your Application" and section 4.6, "WebLogic JMS Clustering".

5.18.10 Message Expiration

Messages can both programmatically and administratively be given expiration times. When a message expires, it either gets deleted or redirected to another destination. This behavior can be used to alleviate overflow conditions by automatically "cleaning up" older messages. Of course, this solution does not work for applications that must operate on every sent message. For more information, see section 5.15, expired message handling.

5.18.11 Consider Message Contents

Consider changing the application to compress its message contents using the standard Java compression libraries built into the JDK. Text and XML messages in particular often compress to a significantly smaller size. Also consider changing the message type, and make sure that messages do not contain significantly more information than is actually needed. For more information, see the *Message Design* section 5.1.

6 Appendix A: Producer Pool Example

The following code sample is for the producer pool example described in section 5.12. It is intended for demonstration purposes, and may be modified freely.

```
import javax.jms.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import weblogic.jms.extensions.WLSession;

/**
 * This example shows a way to create a pool of JMS queue producers.
 * Producers are created on an as needed basis and are cached based on queue
 * JNDI name.  Each producer gets its own jms session
 * and jms connection.  Producers are removed when the pool
 * is closed, or when the JMS provider calls their exception listener.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

public final class SenderPool {

  private class SenderPoolElt implements ExceptionListener {
    String queueName;   // JNDI name of queue
    InitialContext ctx;
    QueueConnectionFactory qcf;
    QueueConnection qc;
    QueueSession qsession;
    QueueSender qsender;
    Queue queue;
    TextMessage textMessage;

    // called by messaging system on connection or session failure
    public void onException(JMSException e) {
      e.printStackTrace();
      remove(this);
      close();
    }

    void close() {
      if (ctx != null) try { ctx.close(); } catch (Exception ignore) {};
      if (qc != null)  try { qc.close();  } catch (Exception ignore) {};
    }
  }

  // a hash map of linked lists of producers that use the same queue
  private HashMap pLists = new HashMap();

  private boolean closed;

  public SenderPool() {
  }

  // get a sender from the pool or create one
  private SenderPoolElt get(String queueName)
  throws JMSException, NamingException {
    synchronized(pLists) {
      LinkedList producers = (LinkedList)pLists.get(queueName);
      if (producers != null && producers.size() > 0)
        return (SenderPoolElt)producers.removeFirst();
    }
    SenderPoolElt spe = new SenderPoolElt();
    try {
      spe.ctx = new InitialContext();
```

```
      spe.qcf = (QueueConnectionFactory)
                  spe.ctx.lookup("javax.jms.QueueConnectionFactory");
      spe.qc = spe.qcf.createQueueConnection();
      spe.qsession = spe.qc.createQueueSession(false, 0);
      spe.queueName = queueName;
      spe.queue = (Queue)spe.ctx.lookup(queueName);
      spe.qsender = spe.qsession.createSender(spe.queue);
      spe.textMessage = spe.qsession.createTextMessage();
      ((WLSession)spe.qsession).setExceptionListener(spe);
      spe.qc.setExceptionListener(spe);
    } catch ( JMSException je ) {
      spe.close();
      throw je;
    } catch ( NamingException ne ) {
      spe.close();
      throw ne;
    }
    return spe;
  }

  // called by spe when it gets an exception
  private void remove(SenderPoolElt spe) {
    synchronized(pLists) {
      LinkedList producers = (LinkedList)pLists.get(spe.queueName);
      if (producers != null) producers.remove(spe);
    }
  }

  /**
   * Send a text message using a producer in the pool
   * that is associated with the given queue.
   */
  public String send(String queueJNDIName, String text, boolean persistent)
  throws JMSException, NamingException {
    synchronized(pLists) {
      if (closed) throw new JMSException("Producer pool closed");
    }

    SenderPoolElt spe;

    try {
      spe = get(queueJNDIName);
    } catch (Exception e) {
      // just for the heck of it, try one more time
      spe = get(queueJNDIName);
    }

    spe.textMessage.clearBody();
    spe.textMessage.setText(text);
    spe.qsender.setDeliveryMode((persistent)?DeliveryMode.PERSISTENT
                                            :DeliveryMode.NON_PERSISTENT);

    try {
      spe.qsender.send(spe.textMessage);
    } catch (JMSException e) {
      spe.close();
      throw e;
    }

    String messageId = spe.textMessage.getJMSMessageID();
    spe.textMessage.clearBody();

    boolean isClosed = false;

    synchronized(pLists) {
      if (closed) {
        // may have closed during send, must clean up spe, but don't
        // want to close it under lock
        isClosed = true;
      } else {
```

```
        // put sender in pool now that we are done
        LinkedList producers = (LinkedList)pLists.get(spe.queueName);
        if (producers == null) {
          producers = new LinkedList();
          pLists.put(spe.queueName, producers);
        }
        producers.add(spe);
      }
    }

    if (isClosed) spe.close();
    return messageId;
  }

  /**
   * Close all producers in the pool.
   */
  public void close() {
    // don't want to call spe.close under a lock, so we use a state flag
    synchronized(pLists) {
      if (closed) return;
      closed = true;
    }
    for (Iterator iter = pLists.values().iterator(); iter.hasNext();) {
      LinkedList producers = (LinkedList)iter.next();
      do {
        if (producers.size() == 0) break;
        ((SenderPoolElt)producers.removeFirst()).close();
      } while (true);
    }
    pLists.clear(); // save the garbage collector some thinking
    pLists = null;
  }
}
```

Here is an example of using pool “mySenderPool” within an asynchronous consumer. It forwards text messages persistently.

```
public void onMessage(Message m) {
  Exception e = null;
  try {
    mySenderPool.send(“MyApplicationQueue”, ((TextMessage)m).getText(), true);
  } catch (NamingException ne) {
    e = ne;
  } catch (JMSException je) {
    e = ne;
  } catch (ClassCastException cce) {
    // expected a text message
    e = cce;
  }
  if (e != null) {
    e.printStackTrace();
    // on MDB:    if tx required, set rollback-only on context
    //            else throw a runtime exception to force a recover
    // on client: if transacted session.rollback()
    //            else session.cleanup()
  }
}
```

7 Appendix B: Approximating Distributed Destinations

7.1 Distributed Queues in 6.1

To distribute a destination across several servers in a cluster, use the distributed destination features built into WebLogic JMS 7.0 and later. If version 7.0 or later is not an option, you can approximate a simple distributed destination when running JMS servers in a "single-tier" configuration. In a single-tier configuration, a local JMS server resides on each server on which a connection factory is targeted.

In a typical scenario, a distributed destination is divided into physical destinations, one physical destination per JMS server. Producers randomly pick which server, and, therefore, which part of the distributed destination where they will produce messages. Meanwhile, consumers (in the form of MDBs) are pinned to a particular physical destination and are replicated homogenously to all physical destinations. To implement this scenario, do the following:

- Create JMS servers on multiple WebLogic servers in the cluster. The JMS servers will collectively host the distributed queue "A". Remember that JMS servers, WebLogic servers, and JMS stores must all be uniquely named within their WebLogic domain.

- Create a queue on each JMS server. These become the physical destinations that collectively become the distributed destination.
    - Give each queue the same name "A".
    - Give each queue the same JNDI name "JNDI_A".
    - Set the queue's "JNDINameReplicated" parameter to false. This parameter is available in versions 8,1, 7.0, 6.1SP4 or later, 6.1SP3 with patch CR081194, and 6.1SP2 with patch CR061106. To configure it, add the following attribute in the queue's config.xml entry:
      JNDINameReplicated="true"
      Alternatively, 8.1 exposes this setting directly on the console.

- Create a connection factory with JNDI name "CF_A", and target it at all servers that have a JMS server with queue "A".

- Target the same MDB pool at each WebLogic server that has a JMS server with queue "A". Configure the MDB's destination to be "JNDI_A". When configuring the MDB, do not specify a URL for a connection factory; the MDB can use the server's default JNDI context, which already contains the destination.

- Producers use connection factory "CF_A" (configured above) to create their connections, and in turn use these connections to create their sessions.

- Producers locate their queue using `javax.jms.QueueSession.createQueue()`. The parameter passed to `createQueue()` requires a special syntax, "./<queue name>". This syntax returns the named destination that is locally hosted on the JMS connection's host server. Therefore, in this example, "./A" returns a physical destination instance of the distributed destination

that is local to the connection's host server. This syntax is supported by 8.1, 7.0, 6.1SP3 or later, and 6.1SP2 with patch CR072612.

- The above two steps are used to ensure that the producer's connection host and target destination are on the same server. Otherwise, the JMS client may end up unintentionally routing its requests through an extra network hop from JMS connection host to JMS server. If this extra hop is not significant relative to overall performance, the JMS client can instead lookup its destination in the standard way using the standard JNDI context (as long as the JNDI context host also hosts one of the desired queues). Alternatively, in WebLogic 8.1, applications can ensure that their JMS connection host and JMS destination host are the same as their JNDI context host by configuring an "affinity-based" load-balancing policy (see the tip under *Connection Load Balancing* in section 4.6.1.5).

This design pattern allows for high availability: if one server goes down, only the messages on that particular server become unavailable, while the distributed destination remains available.

This design pattern also supports high scalability: speedup is directly proportional to the number of servers on which the distributed destination is deployed.

7.2 Distributed Topics in 6.1

To distribute a topic across several servers in a cluster, use the distributed destination features built into WebLogic JMS 7.0 and later. If version 7.0 or later is not an option, you can approximate a simple distributed topic in 6.1 using a method similar to the one described for approximating a distributed queue (see previous section). In addition, you will need to run "forwarder" applications that forward messages between each physical topic instance of the distributed topic. These forwarders are simply topic subscribers that read from each remote local physical topic and forward to the local physical topic. Unlike the internal forwarders used by 7.0 and 8.1 distributed destinations, such forwarders will change the forwarded message's message-id and timestamp, as they are essentially creating a new message when they forward the original; but otherwise, the message will look like an exact replicate.

The forwarder can be implemented via durable subscriber MDBs (sample onMessage() code and descriptor file entries below). Just make sure that each forwarder MDB is located on the same server as the topic it publishes to (this ensures that the publish will succeed), and that a forwarder MDB exists for each remote topic member. The forwarder MDB subscribes to the appropriate remote topic member by configuring the unique URL of the remote host in the WebLogic descriptor file. The MDB container code will automatically try reconnecting the MDB its remote physical topic if the topic's server goes down. The forwarder code should look something like this:

```
void onMessage(javax.jms.Message message) {

  // ensure we don't forward an already forwarded message
  msg.setBooleanProperty("ALREADYFORWARDED", true);

  InitialContext localCtx = new InitialContext();

  // the following default factory will enlist the produce
```

```
    // in the current transaction if one exists
    TopicConnectionFactory topicCF =
      localCtx.lookup("javax.jms.QueueConnectionFactory");

    // this will load-balance to create a local connection,
    // as the topic connection
    // factory should be targeted at every server that hosts this MDB
    TopicConnection topicC = topicCf.createTopicConnection();

    // don't set transacted to true, if you do the publish
    // won't participate in the container transaction
    Topic Session topicS =
       topicC.createSession(false, Session.AUTO_ACKNOWLEDGE);

    Topic topic = localCtx.lookup("myTopicName");
    topicS.createPublisher(topic).publish(msg)

    topicC.close();  // closes underlying session and producer

    localCtx.close();
  }
```

The first line of the onMessage() sets a boolean user property named "forwarded" to guard against forwarding an already forwarded message. To take advantage of this, the MDB which does the forwarding must additionally configure a JMS selector to filter out already forwarded messages "`NOT ALREADY FORWARDED`".

To improve performance of the MDB, you can use producer pooling (see section 5.12) or cache the JMS objects in the MDB state. Another way to improve performance is to configure each MDB pool to have multiple instances by setting the MDB's `max-beans-in-free-pool` descriptor setting greater than one. However, this will have the side effect of changing the order of forwarded messages.

Too ensure "exactly-once" forwarding, ensure that the MDB is transactional by setting the MDB's descriptor transaction settings to "Container" and "Required".

Here is sample XML snippet for the MDB's standard EJB deployment descriptor that ensures "exactly-once" forwarding, ensures that the forwarder does not forward already forwarded messages via a selector, and ensures that the forwarder does not lose messages when it is not booted. If "exactly-once" forwarding is not required do not set the `transaction-type` to `Container` and do not set `trans-attribute` to `Required`. If it is OK to lose forwarded messages due to either the local or the remote side failures, do not set `subscription-durability` to `Durable`. Exactly-once (transactions) and subscription durability (persistence) both incur performance penalties.

```
<ejb-jar>

<enterprise-beans>
  <message-driven>
    <ejb-name>MessageForwarderA</ejb-name>
    <ejb-class>MessageForwarder</ejb-class>
    <transaction-type>Container</transaction-type>
    <message-selector>NOT ALREADYFORWARDED</message-selector>
```

```
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
        <subscription-durability>Durable</subscription-durability>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>MessageForwarderA</ejb-name>
        <method-name>onMessage()</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

  </ejb-jar>
```

Here is sample XML for the MDB's WebLogic deployment descriptor. It specifies the exact URL of the remote physical topic in order to ensure that the MDB is communicating with correct physical topic instance. It also sets `max-beans-in-free-pool` to one in order to ensure messages are forwarded in order.

```
  <ejb-name>MessageForwarderA</ejb-name>
  <message-driven-descriptor>
    <pool>
      <max-beans-in-free-pool>1</max-beans-in-free-pool>
    </pool>
    <provider-url>t3://localhost:10001</provider-url>
    <destination-jndi-name>myTopicName</destination-jndi-name>
  </message-driven-descriptor>
```

If exactly-once and ordered forwarding requires higher performance than an MDB can supply, then a customer forwarder can be written that is launched via a WebLogic startup class. Such a forwarder could batch several subscribes and publishes under the same transaction (as the messaging bridge does).