Red Hat JBoss A-MQ 6.3

Tuning Guide

Optimize Red Hat JBoss A-MQ for your environment

Last Updated: 2019-08-16

Red Hat JBoss A-MQ 6.3 Tuning Guide

Optimize Red Hat JBoss A-MQ for your environment

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide describes many of the tricks that can be used to fine tune a broker instance.

Table of Contents

CHAPTER 1. INTRODUCTION TO PERFORMANCE TUNING

LIMITING FACTORS

Before considering how to improve performance, you need to consider what the limiting factors are. The main obstacles to a fast messaging system are, as follows:

- The speed at which messages are written to and read from disk (persistent brokers only).
- The speed at which messages can be marshalled and sent over the network.
- Context switching, due to multi-threading.

Ultimately, the tuning suggestions in this guide tackle the preceding limits in various ways.

NON-PERSISTENT AND PERSISTENT BROKERS

The range of options for tuning non-persistent brokers are slightly different from the options for tuning persistent brokers. Most of the tuning techniques described in this chapter can be applied either to non-persistent or persistent brokers (with the exception of flow control, which is only relevant to non-persistent brokers).

Techniques specific to persistent brokers are discussed in Chapter 6, *Persistent Messaging*.

BROKER NETWORKS

One of the major techniques for coping with large scale messaging systems is to establish a broker network, with brokers deployed on multiple hosts. This topic is discussed briefly in Section 2.7, “Horizontal Scaling”, but for a comprehensive discussion and explanation of how to set up a broker network, please consult *Using Networks of Brokers*.

CHAPTER 2. GENERAL TUNING TECHNIQUES

Abstract

This chapter outlines the tuning techniques that can be used to optimize the performance of either a non-persistent broker or a persistent broker.

2.1. SYSTEM ENVIRONMENT

Overview

Before discussing how to tune the performance of a Red Hat JBoss A-MQ application, it is worth recalling that performance is also affected by the system environment.

Disk speed

For persistent brokers, disk speed is a significant factor affecting performance. For example, whereas the typical seek time for an ordinary desktop drive is 9ms, the seek time of a high-end server disk could be as little as 3ms. You should also ensure that disks do not become excessively fragmented.

Network performance

For both persistent and non-persistent brokers, the network speed can be a limiting factor. Evidently, there are limits to what can be achieved through JBoss A-MQ tuning, if the underlying network is very slow. One strategy that you can try is to enable compression of large messages (see the section called "Enabling compression"). In this case, it is also important to avoid delays caused by latency; for this reason, it might be a good idea to enable more asynchronous behaviour.

Hardware specification

Relevant aspects of the underlying hardware include the speed and number of CPUs, and the memory available to the broker. In particular, increasing the available memory can bring several performance advantages.

For example, if the broker's entire B-tree message index can fit into memory, this significantly reduces the amount of reading and writing to disk that is required. Also, if some consumers are slow, causing messages to back up in the broker, it can be an advantage to have a large amount of memory available to buffer the pending messages.

Memory available to the JVM

To increase the amount of memory available to a JVM instance, use the **-Xmx** option. For example, to increase JVM memory to 2048 MB, add **-Xmx2048M** (or equivalently, **-Xmx2G**) as a JVM option.

2.2. CO-LOCATING THE BROKER

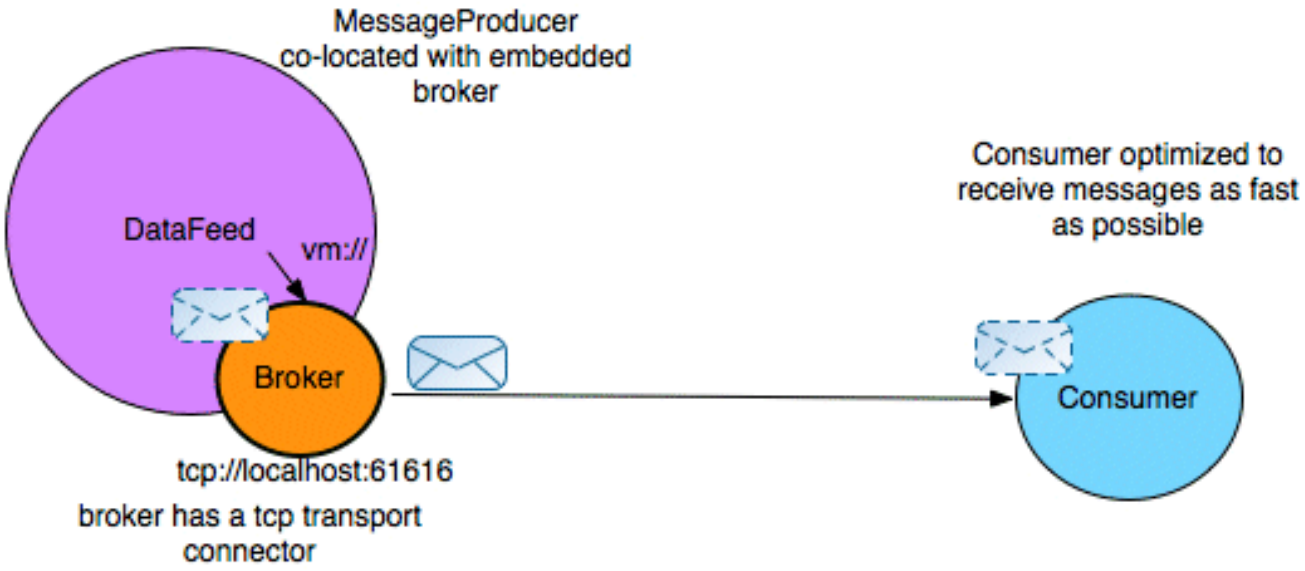
Overview

An obvious way to improve network performance is to eliminate one of the hops in the messaging application. With a standalone broker, at least two hops are required to route a message from producer to consumer: the *producer-to-broker* hop and the *broker-to-consumer* hop. On the other hand, by

embedding the broker (either in the producer or in the consumer), it is possible to eliminate one of the hops, thereby halving the load on the network.

Figure 2.1, "Broker Co-located with Producer" shows an example of a data feed that acts as a message producer, sending a high volume of messages through the broker. In this case, it makes perfect sense for the broker to be co-located with the data feed, so that messages can be sent directly to the consumers, without the need for an intermediate hop. The simplest way to create an embedded broker is to exploit Red Hat JBoss A-MQ's **vm:**// transport.

Figure 2.1. Broker Co-located with Producer



The vm:// transport

You can connect to a **vm:**// endpoint from a producer or a consumer in just the same way as you connect to a **tcp:**// endpoint (or any other protocol supported by Red Hat JBoss A-MQ). But the effect of connecting to a **vm:**// endpoint is quite different from connecting to a **tcp:**// endpoint: whereas a **tcp:**// endpoint initiates a connection to a remote broker instance, the **vm:**// endpoint actually creates a *local, embedded broker instance*. The embedded broker runs inside the same JVM as the client and messages are sent to the broker through an internal channel, bypassing the network.

For example, an Apache Camel client can create a simple, embedded broker instance by connecting to a URL of the following form:

```
vm://brokerName
```

Where *brokerName* uniquely identifies the embedded broker instance. This URL creates a simple broker instance with a default configuration. If you want to define the broker configuration precisely, however, the most convenient approach is to specify a broker configuration file, by setting the **brokerConfig** option. For example, to create a **myBroker** instance that takes its configuration from the **activemq.xml** configuration file, define the following VM endpoint:

```
vm://myBroker?brokerConfig=xbean:activemq.xml
```

For more details, see the *Connection Reference*.

A simple optimization

By default, the embedded broker operates in asynchronous mode, so that calls to a send method return

immediately (in other words, messages are dispatched to consumers in a separate thread). If you turn off asynchronous mode, however, you can reduce the amount of context switching. For example, you can disable asynchronous mode on a VM endpoint as follows:

```
vm://brokerName?async=false
```

NOTE

If both the broker option, **optimizedDispatch**, and the consumer option, **dispatchAsync**, are also configured to disable asynchronous behaviour, the calling thread can actually dispatch directly to consumers.

2.3. OPTIMIZING THE PROTOCOLS

Overview

Protocol optimizations can be made in different protocol layers, as follows:

- the section called "TCP transport" .
- the section called "OpenWire protocol" .
- the section called "Enabling compression" .

TCP transport

In general, it is usually possible to improve the performance of the TCP layer by increasing buffer sizes, as follows:

- *Socket buffer size*—the default TCP socket buffer size is 64 KB. While this is adequate for the speed of networks in use at the time TCP was originally designed, this buffer size is sub-optimal for modern high-speed networks. The following rule of thumb can be used to estimate the optimal TCP socket buffer size:

  Buffer Size = Bandwidth x Round-Trip-Time

  Where the *Round-Trip-Time* is the time between initially sending a TCP packet and receiving an acknowledgement of that packet (ping time). Typically, it is a good idea to try doubling the socket buffer size to 128 KB. For example:

  ```
  tcp://hostA:61617?socketBufferSize=131072
  ```

  For more details, see the Wikipedia article on Network Improvement.

- *I/O buffer size*—the I/O buffer is used to buffer the data flowing between the TCP layer and the protocol that is layered above it (such as OpenWire). The default I/O buffer size is 8 KB and you could try doubling this size to achieve better performance. For example:

  ```
  tcp://hostA:61617?ioBufferSize=16384
  ```

OpenWire protocol

The OpenWire protocol exposes several options that can affect performance, as shown in Table 2.1, "OpenWire Parameters Affecting Performance".

Table 2.1. OpenWire Parameters Affecting Performance

Parameter	Default	Description
cacheEnabled	**true**	Specifies whether to cache commonly repeated values, in order to optimize marshaling.
cacheSize	**1024**	The number of values to cache. Increase this value to improve performance of marshaling.
tcpNoDelayEnabled	**false**	When **true**, disable the Nagles algorithm. The Nagles algorithm was devised to avoid sending tiny TCP packets containing only one or two bytes of data; for example, when TCP is used with the Telnet protocol. If you disable the Nagles algorithm, packets can be sent more promptly, but there is a risk that the number of very small packets will increase.
tightEncodingEnabled	**true**	When **true**, implement a more compact encoding of basic data types. This results in smaller messages and better network performance, but comes at a cost of more calculation and demands made on CPU time. A trade off is therefore required: you need to determine whether the network or the CPU is the main factor that limits performance.

To set any of these options on an Apache Camel URI, you must add the **wireFormat.** prefix. For example, to double the size of the OpenWire cache, you can specify the cache size on a URI as follows:

```
tcp://hostA:61617?wireFormat.cacheSize=2048
```

Enabling compression

If your application sends large messages and you know that your network is slow, it might be worthwhile to enable compression on your connections. When compression is enabled, the body of each JMS message (but not the headers) is compressed before it is sent across the wire. This results in smaller messages and better network performance. On the other hand, it has the disadvantage of being CPU intensive.

To enable compression, enable the **useCompression** option on the **ActiveMQConnectionFactory** class. For example, to initialize a JMS connection with compression enabled in a Java client, insert the following code:

```
// Java
...
// Create the connection.
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(user, password,
url);
connectionFactory.setUseCompression(true);
Connection connection = connectionFactory.createConnection();
connection.start();
```

Alternatively, you can enable compression by setting the **jms.useCompression** option on a producer URI—for example:

```
tcp://hostA:61617?jms.useCompression=true
```

2.4. MESSAGE ENCODING

Message body type

JMS defines five message body types:

- **StreamMessage**
- **MapMessage**
- **TextMessage**
- **ObjectMessage**
- **BytesMessage**

Of these message types, **BytesMessage** (a stream of uninterpreted bytes) is the *fastest*, while **ObjectMessage** (serialization of a Java object) is the *slowest*.

Encoding recommendation

For best performance, therefore, it is recommended that you use **BytesMessage** whenever possible. We suggest that you use Google's Protobuf, which has excellent performance characteristics.

2.5. THREADING OPTIMIZATIONS

Optimized dispatch

On the broker, you can reduce the number of required threads by setting the **optimizedDispatch** option to **true** on all queue destinations. When this option is enabled, the broker no longer uses a dedicated thread to dispatch messages to each destination.

For example, to enable the **optimizedDispatch** option on all queue destinations, insert the following policy entry into the broker configuration:

```
<broker ... >
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry queue=">" optimizedDispatch=”true” />
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

Where the value of the **queue** attribute, **>**, is a wildcard that matches all queue names.

2.6. VERTICAL SCALING

Definition

Vertical scaling refers to the capacity of a single broker to support large numbers of connections from consumers and producers.

Tricks to optimize vertical scaling

You can exploit the following tricks to optimize vertical scaling in Red Hat JBoss A-MQ:

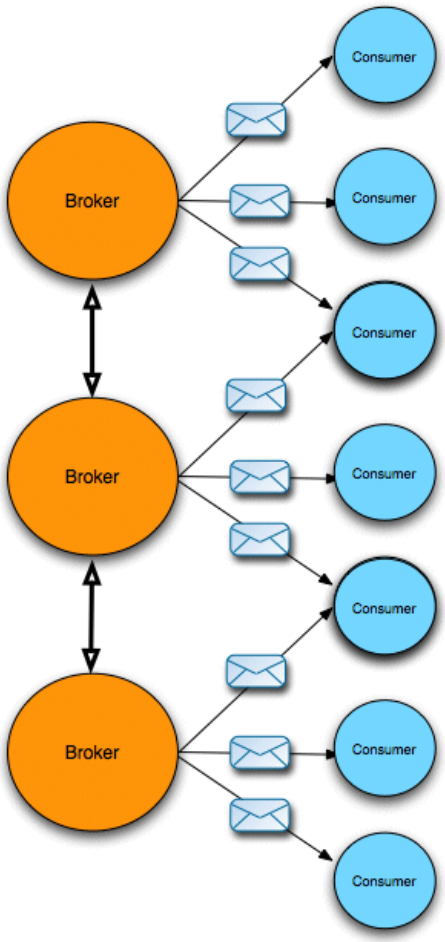
- *NIO transport on the broker*—to reduce the number of threads required, use the NIO transport (instead of the TCP transport) when defining transport connectors in the broker. *Do not use the NIO transport in clients*, it is only meant to be used in the broker.
- *Allocate more memory to broker*—to increase the amount of memory available to the broker, pass the **-Xmx** option to the JVM.
- *Reduce initial thread stack size*—to allocate a smaller initial stack size for threads, pass the **-Xss** option to the JVM.

2.7. HORIZONTAL SCALING

Overview

Horizontal scaling refers to the strategy of increasing capacity by adding multiple brokers to your messaging network. Figure 2.2, “Scaling with Multiple Brokers” illustrates how a broker network can be used to support a large number of consumers.

Figure 2.2. Scaling with Multiple Brokers



Broker networks

You can improve the scalability of your messaging system by adding multiple brokers to the system, thus escaping the inherent resource limits of a broker deployed on a single machine. Brokers can be combined into a network by adding *network connectors* between the brokers, which enables you to define broker networks with an arbitrary topology.

When brokers are linked together as a network, routes from producers to consumers are created dynamically, as clients connect to and disconnect from the network. That is, with the appropriate topology, a consumer can connect to any broker in the network and the network automatically routes messages from producers attached at any other point in the network.

For a detailed discussion of how to set up broker networks, see "Using Networks of Brokers" .

Static scales better than dynamic

Red Hat JBoss A-MQ offers two alternative strategies for routing messages through a broker network: *static propagation* and *dynamic propagation*.

Although dynamic propagation is more flexible, it necessitates sending advisory messages throughout the broker network, which the brokers then use to figure out the optimal route in a dynamic way. As you scale up the network, there is a danger that the advisory messages could swamp the traffic in the broker network.

Static propagation requires you to specify routes explicitly, by telling the broker where to forward messages for specific queues and topics (you can use pattern matching). In this case, you can configure

the brokers to disable advisory messages altogether, which eliminates the scalability problems associated with advisory messages.

Asynchronous network connection establishment

By default, a broker establishes connections to its peers in the network using a single thread. If the broker connects to a large number of peers, however, the single-threaded approach can result in a very slow broker start-up time. For example, if one or more of the peers responds slowly, the initiating broker has to wait until that slow connection is established before proceeding to establish the remaining connections.

In this case, you can accelerate the broker start-up by enabling *asynchronous* network connection establishment. This feature employs a thread pool to establish network connections in parallel. Set the **networkConnectorStartAsync** attribute on the **broker** element to **true**, as follows:

```
<beans ...>
  <broker ... networkConnectorStartAsync="true">...</broker>
</beans>
```

Client-side traffic partitioning

An alternative horizontal scaling strategy is to deploy multiple brokers, but to leave them isolated from one another, so that there is no broker network. In this case, you can leave it up to the clients to decide which broker to send messages to or receive messages from. This requires messages to be partitioned into different categories (for example, based on the destination name), so that specific message categories are sent to a particular broker.

The advantages of this approach are:

- You can use all the tuning techniques for vertical scaling.
- You can achieve better horizontal scalability than a network of brokers (because there is less broker crosstalk).

The disadvantage of this approach is that your clients are slightly more complex, because they must implement the partitioning of messages into different categories and the selection of the appropriate broker.

2.8. INTEGRATION WITH SPRING AND CAMEL

Overview

Spring supports a useful abstraction, **JmsTemplate**, which allows you to hide some of the lower level JMS details when sending messages and so on. One thing to bear in mind about **JmsTemplate**, however, is that it creates a new connection, session, and producer for every message it sends, which is very inefficient. It is implemented like this in order to work inside an EJB container, which typically provides a special JMS connection factory that supports connection pooling.

If you are not using an ESB container to manage your JMS connections, we recommend that you use the pooling JMS connection provider, **org.apache.activemq.pool.PooledConnectionFactory**, from the **activemq-pool** artifact, which pools JMS resources to work efficiently with Spring's **JmsTemplate** or with EJBs.

Creating a pooled connection factory

The **PooledConnectionFactory** is implemented as a wrapper class that is meant to be chained with another connection factory instance. For example, you could use a **PooledConnectionFactory** instance to wrap a plain Red Hat JBoss A-MQ connection factory, or to wrap an **ActiveMQSslConnectionFactory**, and so on.

Example

For example, to instantiate a pooled connection factory, **jmsFactory**, that works efficiently with the Spring **JmsTemplate** instance, **myJmsTemplate**, define the following bean instances in your Spring configuration file:

```
<!-- A pooling-based JMS provider -->
<bean id="jmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" destroy-
method="stop">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL">
        <value>tcp://localhost:61616</value>
      </property>
    </bean>
  </property>
</bean>

<!-- Spring JMS Template -->
<bean id="myJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory">
    <ref local="jmsFactory"/>
  </property>
</bean>
```

In the preceding example, the pooled connection factory is chained with a plain **ActiveMQConnectionFactory** instance that opens connections to the **tcp://localhost:61616** broker endpoint.

2.9. OPTIMIZING MEMORY USAGE IN THE BROKER

Optimize message paging

By setting the page size attributes on the **policyEntry** element, you can tune the message paging to match the amount of memory available in the broker. For example, if there is very large queue and lots of destination memory, increasing the **maxBrowsePage** attribute would allow more of those messages to be visible when browsing a queue.

Destination policies to control paging

The following destination policies control message paging (the number of messages that are pulled into memory from the message store, each time the memory is emptied):

maxPageSize

The maximum number of messages paged into memory for sending to a destination.

maxBrowsePageSize

The maximum number of messages paged into memory for browsing a queue .

NOTE

The number of messages paged in for browsing cannot exceed the destination's **memoryLimit** setting.

maxExpirePageSize

The maximum number of messages paged into memory to check for expired messages.

Automatic adjustment of store usage limit

If the broker is started and the amount of available space is less than the **storeUsage** limit, the runtime store usage limit is reset automatically and the broker starts. If the available disk space is very small, however, this can lead to a broker starting that cannot process messages. To avoid this happening, you can set the **adjustUsageLimits** attribute to **false** on the **broker** element, which ensures that the broker does *not* start if the available disk space is less than the **storeUsage** limit.

For example, you can set the **adjustUsageLimits** attribute as follows:

```
<broker adjustUsageLimits="false" ...> ... </broker>
```

2.10. OPTIMIZING JMX MANAGEMENT

Selective MBean registration

In situations where you need to scale your broker to a large number of connections, destinations, and consumers it can become very expensive to keep JMX MBeans for all those objects. Instead of turning off JMX completely, however, you can selectively suppress registration of some types of MBeans and thus help your broker to scale, while still having a basic view of the broker state.

For example, the following configuration excludes all dynamic producers, consumers, connections and advisory topics, preventing them from registering their MBeans:

```
<managementContext>
<managementContext

suppressMBean="endpoint=dynamicProducer,endpoint=Consumer,connectionName=*,destinationName
=ActiveMQ.Advisory.*"
>
</managementContext>
```

CHAPTER 3. CONSUMER PERFORMANCE

3.1. ACKNOWLEDGEMENT MODES

Overview

The choice of *acknowledgement mode* in a consumer has a large impact on performance, because each acknowledgement message incurs the overhead of a message send over the network. The key to improving performance in this area is to send acknowledgements *in batches*, rather than acknowledging each message individually.

Supported acknowledgement modes

Red Hat JBoss A-MQ supports the following acknowledgement modes:

Session.AUTO_ACKNOWLEDGE

(Default) In this mode, the JMS session automatically acknowledges messages as soon as they are received. In particular, the JMS session acknowledges messages *before* dispatching them to the application layer. For example, if the consumer application calls **MessageConsumer.receive()**, the message has already been acknowledged before the call returns.

Session.CLIENT_ACKNOWLEDGE

In this mode, the client application code explicitly calls the **Message.acknowledge()** method to acknowledge the message. In Apache Camel, this acknowledges not just the message on which it is invoked, but also any other messages in the consumer that have already been completely processed.

Session.DUPS_OK_ACKNOWLEDGE

In this mode, the JMS session automatically acknowledges messages, but does so in a lazy manner. If JMS fails while this mode is used, some messages that were completely processed could remain unacknowledged. When JMS is restarted, these messages will be re-sent (duplicate messages).

This is one of the fastest acknowledgement modes, but the consumer must be able to cope with possible duplicate messages (for example, by detecting and discarding duplicates).

Session.SESSION_TRANSACTED

When using transactions, the session implicitly works in **SESSION_TRANSACTED** mode. The response to the transaction commit is then equivalent to message acknowledgement.

When JMS transactions are used to group multiple messages, transaction mode is very efficient. But avoid using a transaction to send a single message, because this incurs the extra overhead of committing or rolling back the transaction.

ActiveMQSession.INDIVIDUAL_ACKNOWLEDGE

This non-standard mode is similar to **CLIENT_ACKNOWLEDGE**, except that it acknowledges *only* the message on which it is invoked. It does not flush acknowledgements for any other completed messages.

optimizeAcknowledge option

The **optimizeAcknowledge** option is exposed on the **ActiveMQConnectionFactory** class and must be

used in conjunction with the **Session.AUTO_ACKNOWLEDGE** mode. When set to **true**, the consumer acknowledges receipt of messages in batches, where the batch size is set to 65% of the prefetch limit. Alternatively, if message consumption is slow, the batch acknowledgement will be sent after 300ms. Default is **false**.

You can set this option on a consumer URI, as follows:

```
tcp://hostA:61617?jms.optimizeAcknowledge=true
```

NOTE

The **optimizeAcknowledge** option is only supported by the JMS client API.

Choosing the acknowledgement mode

In general, you can achieve the best performance either using JMS transactions, and grouping several messages into a single transaction, or by selecting the **DUPS_OK_ACKNOWLEDGE** mode, which requires you to implement duplicate detection code in your consumer.

A typical strategy for implementing duplicate detection is to insert a unique message ID in a JMS header on the producer side and then to store the received IDs on the consumer side. If you are using Red Hat JBoss A-MQ in combination with Apache Camel, you can easily use the Idempotent Consumer pattern to implement duplicate detection.

3.2. REDUCING CONTEXT SWITCHING

Overview

Through the consumer configuration options, there are two different ways in which you can optimize the threading model:

- the section called “Optimize message dispatching on the broker side” .
- the section called “Optimize message reception on the consumer side” .

Optimize message dispatching on the broker side

On the broker side, the broker normally dispatches messages to consumers asynchronously, which usually gives the best performance (that is, it enables the broker to cope better with slow consumers). If you are sure that your consumers are always fast, however, you could achieve better performance by disabling asynchronous dispatch on the broker (thereby avoiding the cost of unnecessary context switching).

Broker-side asynchronous dispatching can be enabled or disabled at the granularity of individual consumers. Hence, you can disable asynchronous dispatching for your fast consumers, but leave it enabled for your (possibly) slow consumers.

To disable broker-side asynchronous dispatching, set the **consumer.dispatchAsync** option to **false** on the transport URI used by the consumer. For example, to disable asynchronous dispatch to the **TEST.QUEUE** queue, use the following URI on the consumer side:

```
TEST.QUEUE?consumer.dispatchAsync=false
```

It is also possible to disable asynchronous dispatch by setting the **dispatchAsync** property to false on the ActiveMQ connection factory—for example:

```
// Java
((ActiveMQConnectionFactory)connectionFactory).setDispatchAsync(false);
```

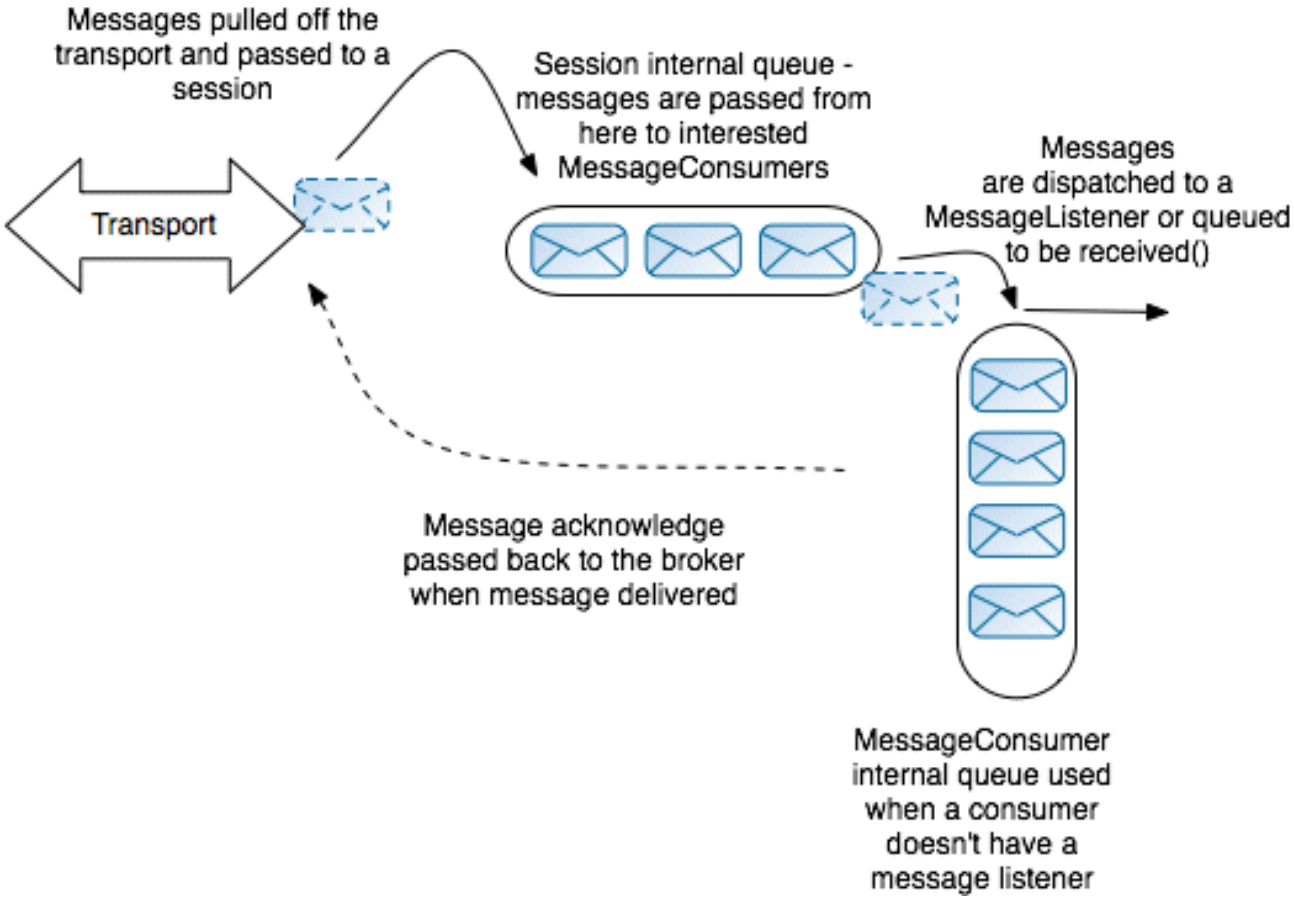
Optimize message reception on the consumer side

On the consumer side, there are two layers of threads responsible for receiving incoming messages: the **Session** threads and the **MessageConsumer** threads. In the special case where only one session is associated with a connection, the two layers are redundant and it is possible to optimize the threading model by eliminating the thread associated with the session layer. This section explains how to enable this consumer threading optimization.

Default consumer threading model

Figure 3.1, "Default Consumer Threading Model" gives an overview of the default threading model on a consumer. The first thread layer is responsible for pulling messages directly from the transport layer, marshalling each message, and inserting the message into a queue inside a **javax.jms.Session** instance. The second thread layer consists of a pool of threads, where each thread is associated with a **javax.jms.MessageConsumer** instance. Each thread in this layer picks the relevant messages out of the session queue, inserting each message into a queue inside the **javax.jms.MessageConsumer** instance.

Figure 3.1. Default Consumer Threading Model

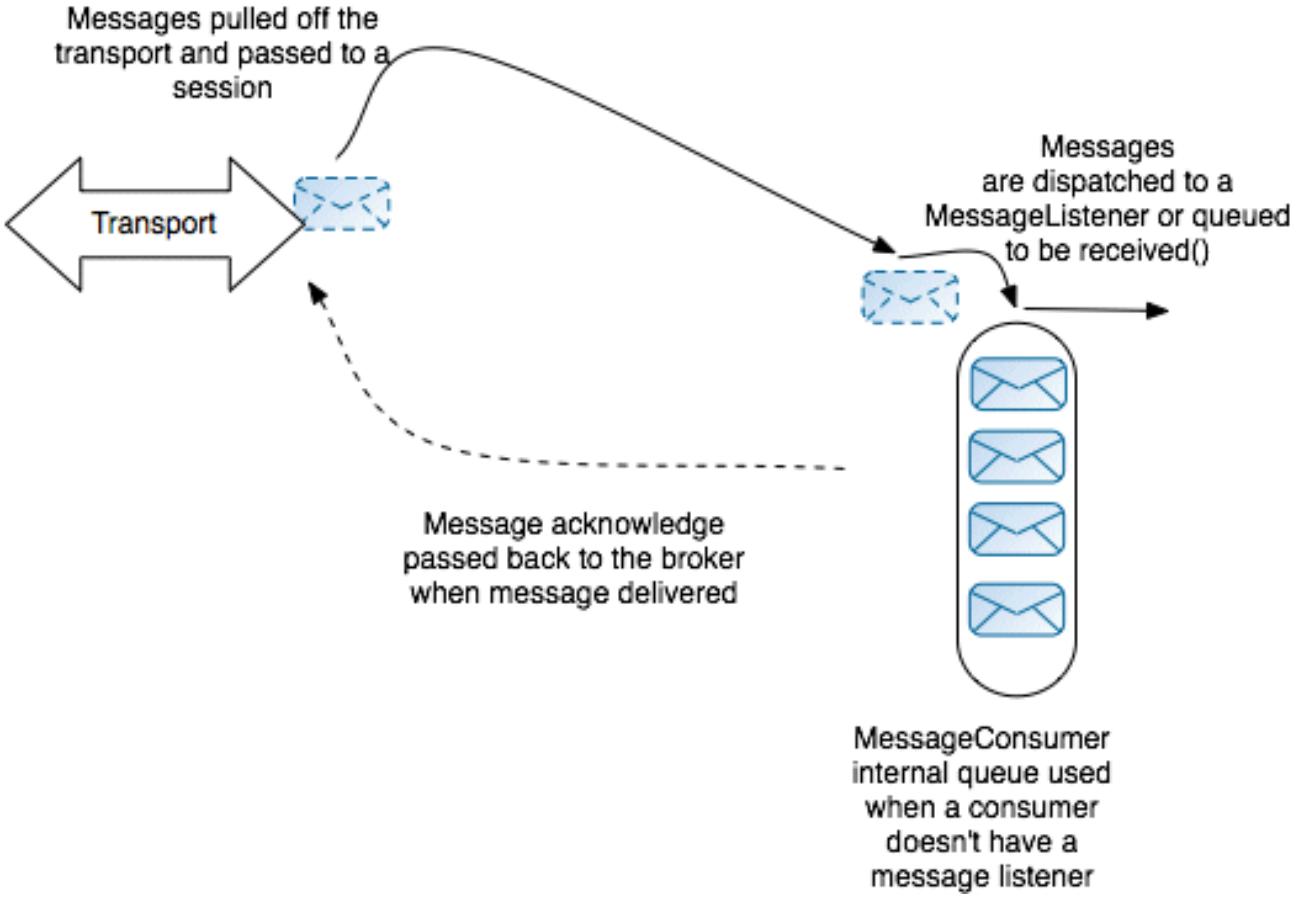


Optimized consumer threading model

Figure 3.2, "Optimized Consumer Threading Model" gives an overview of the optimized consumer threading model. This threading model can be enabled, only if there is *no more than one session*

associated with the connection. In this case, it is possible to optimize away the session threading layer and the **MessageConsumer** threads can then pull messages directly from the transport layer.

Figure 3.2. Optimized Consumer Threading Model



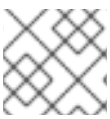
Prerequisites

This threading optimization only works, if the following prerequisites are satisfied:

1. There must only be one JMS session on the connection. If there is more than one session, a separate thread is always used for each session, irrespective of the value of the **alwaysSessionAsync** flag.
2. One of the following acknowledgement modes must be selected:
   - **Session.DUPS_OK_ACKNOWLEDGE**
   - **Session.AUTO_ACKNOWLEDGE**

alwaysSessionAsync option

To enable the consumer threading optimization, set the **alwaysSessionAsync** option to **false** on the **ActiveMQConnectionFactory** (default is **true**).



NOTE

The **optimizeAcknowledge** option is only supported by the JMS client API.

Example

The following example shows how to initialize a JMS connection and session on a consumer that exploits the threading optimization by switching off the **alwaysSessionAsync** flag:

```
// Java
...
// Create the connection.
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(user, password,
url);
connectionFactory.setAlwaysSessionAsync(false);
Connection connection = connectionFactory.createConnection();
connection.start();

// Create the one-and-only session on this connection.
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

3.3. PREFETCH LIMIT

Overview

If a consumer is slow to acknowledge messages, it can happen that the broker sends it another message before the previous message is acknowledged. If the consumer continues to be slow, moreover, the number of unacknowledged messages can grow continuously larger. The broker does not continue to send messages indefinitely in these circumstances. When the number of unacknowledged messages reaches a set limit—the *prefetch limit*—the server ceases sending new messages to the consumer. No more messages will be sent until the consumer starts sending back some acknowledgements.

Prefetch limits

Different prefetch limits can be set for each consumer type:

Queue consumer

Default prefetch limit is 1000.

If you are using a collection of consumers to distribute the workload (many consumers processing messages from the same queue), you typically want this limit to be small. If one consumer is allowed to accumulate a large number of unacknowledged messages, it could starve the other consumers of messages. Also, if the consumer fails, there would be a large number of messages unavailable for processing until the failed consumer is restored.

Queue browser

Default prefetch limit is 500.

Topic consumer

Default prefetch limit is 32766.

The default limit of 32766 is the largest value of a short and is the maximum possible value of the prefetch limit.

Durable topic subscriber

Default prefetch limit is 100.

You can typically improve the efficiency of a consumer by increasing this prefetch limit.

Optimizing prefetch limits

Typically, it is a good idea to optimize queue consumers and durable topic subscribers as follows:

- *Queue consumers*—if you have just a single consumer attached to a queue, you can leave the prefetch limit at a fairly large value. But if you are using a group of consumers to distribute the workload, it is usually better to restrict the prefetch limit to a very small number—for example, 0 or 1.

- *Durable topic subscribers*—the efficiency of topic subscribers is generally improved by increasing the prefetch limit. Try increasing the limit to 1000.

CHAPTER 4. PRODUCER PERFORMANCE

4.1. ASYNC SENDS

Overview

ActiveMQ supports sending messages to a broker in either *synchronous* or *asynchronous* mode. The selected mode has a large impact on the latency of the send call: synchronous mode increases latency and can lead to a reduction in the producer's throughput; asynchronous mode generally improves throughput, but it also affects reliability.

Red Hat JBoss A-MQ sends messages in asynchronous mode by default in several cases. It is only in those cases where the JMS specification *requires* the use of synchronous mode that the producer defaults to synchronous sending. In particular, JMS requires synchronous sends when persistent messages are being sent outside of a transaction.

If you are not using transactions and are sending persistent messages, each send is synchronous and blocks until the broker has sent an acknowledgement back to the producer to confirm that the message is safely persisted to disk. This acknowledgement guarantees that the message will not be lost, but it also has a large latency cost.

Many high performance applications are designed to tolerate a small amount of message loss in failure scenarios. If your application is designed in this fashion, you can enable the use of *async sends* to increase throughput, even when using persistent messages.

Configuring on a transport URI

To enable async sends at the granularity level of a single producer, set the **jms.useAsyncSend** option to **true** on the transport URI that you use to connect to the broker. For example:

```
tcp://locahost:61616?jms.useAsyncSend=true
```

Configuring on a connection factory

To enable async sends at the granularity level of a connection factory, set the **useAsyncSend** property to **true** directly on the **ActiveMQConnectionFactory** instance. For example:

```
// Java
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

Configuring on a connection

To enable async sends at the granularity level of a JMS connection, set the **useAsyncSend** property to **true** directly on the **ActiveMQConnection** instance. For example:

```
// Java
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

4.2. FLOW CONTROL

Overview

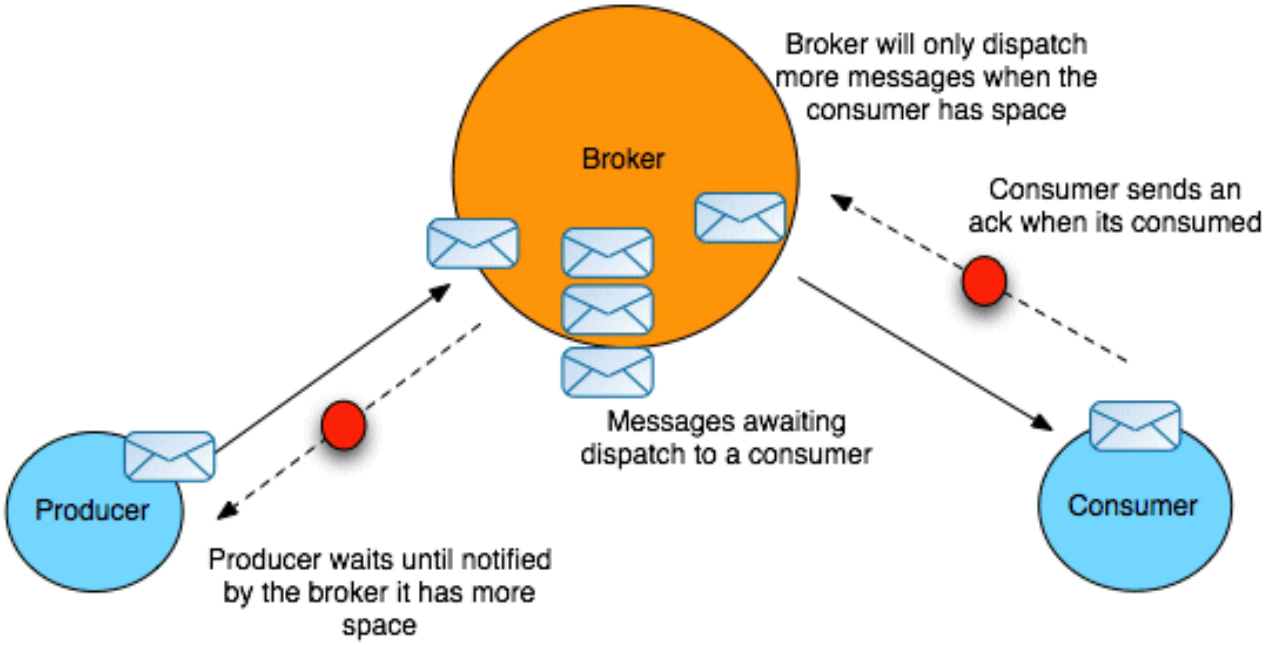
Memory limits, when configured, prevent the broker from running out of resources. The default behaviour, when a limit is reached, is to block the sending thread in the broker, which blocks the destination and the connection.

Producer flow control is a mechanism that pushes the blocking behaviour onto the client, so that the producer thread blocks if the broker has no space. With producer flow control, the producer has a send window that is dependent on broker memory. When the send window is full, it blocks on the client.

Flow control enabled

Figure 4.1, "Broker with Flow Control Enabled" gives an overview of what happens to a messaging application when flow control is enabled.

Figure 4.1. Broker with Flow Control Enabled



If a consumer is very slow at acknowledging messages (or stops acknowledging messages altogether), the broker continues to dispatch messages to the consumer until it reaches the prefetch limit, after which the messages start to back up on the broker. Assuming the producer continues to produce lots of messages and the consumer continues to be very slow, the broker will start to run short of memory resources as it holds on to pending messages for the consumer.

When the consumed memory resources start to approach their limit (as defined either by the per-destination memory limit or the per-broker memory limit), the flow control mechanism activates automatically in order to protect the broker resources. The broker sends a message to the producer asking it either to slow down or to stop sending messages to the broker. This protects the broker from running out of memory (and other) resources.

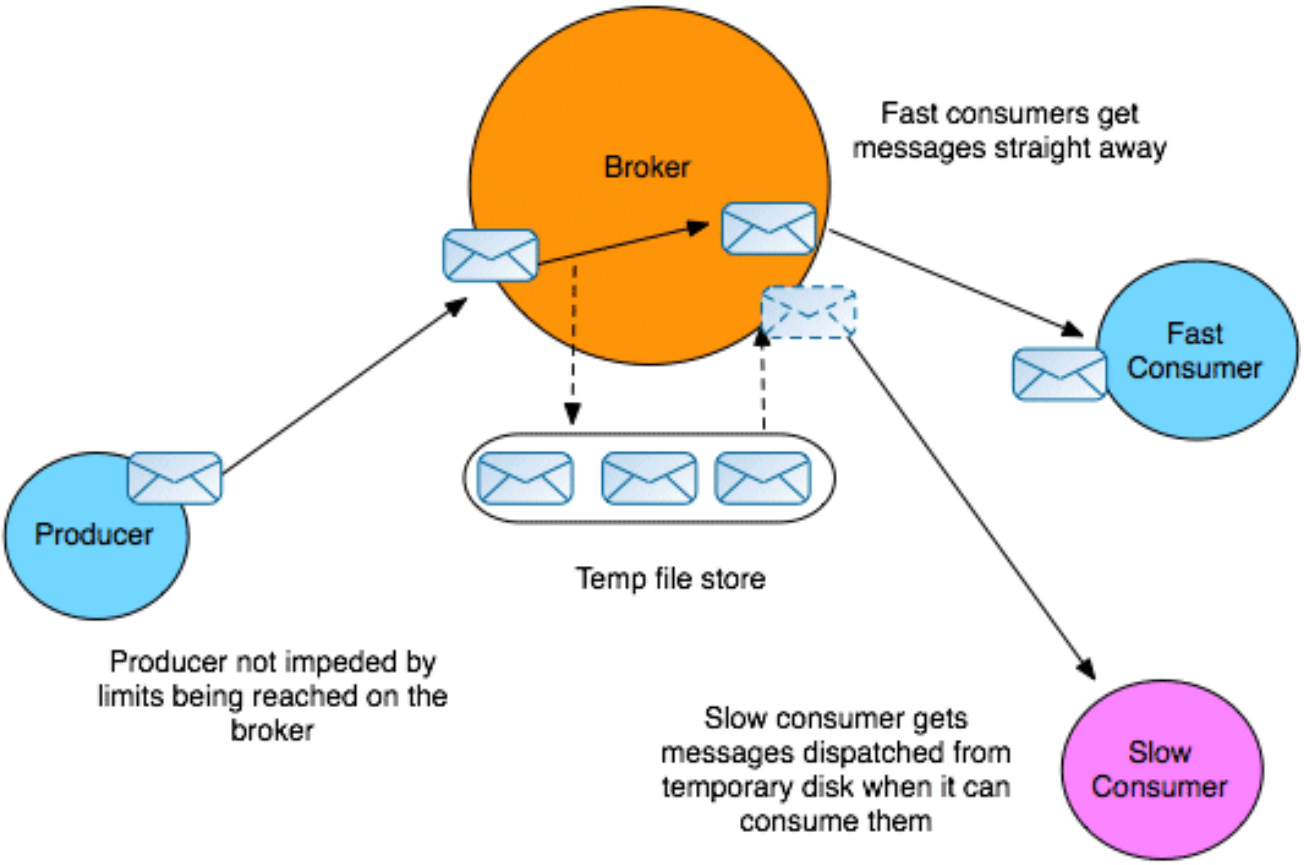
NOTE

There are some differences in behaviour between a persistent broker and a non-persistent broker. If a broker is *persistent*, pending messages are stored to disk, but flow control can still be triggered if the amount of memory used by a cursor approaches its limit (see Section 6.3, "vmCursor on Destination" for more details about cursors).

Flow control disabled

While it is generally a good idea to enable flow control in a broker, there are some scenarios for which it is unsuitable. Consider the scenario where a producer dispatches messages that are consumed by multiple consumers (for example, consuming from a topic), but one of the consumers could fail without the broker becoming aware of it right away. This scenario is shown in Figure 4.2, “Broker with Flow Control Disabled”.

Figure 4.2. Broker with Flow Control Disabled



Because the slow consumer remains blocked for a very long time (possibly indefinitely), after flow control kicks in, the producer also ceases producing messages for a very long time (possibly indefinitely). This is an undesirable outcome, because there are other active consumers interested in the messages coming from the producer and they are now being unfairly deprived of those messages.

In this case, it is better to turn off flow control in the broker, so that the producer can continue sending messages to the other interested consumers. The broker now resorts to an alternative strategy to avoid running out of memory: the broker writes any pending messages for the slow consumer to a temporary file. Ultimately, this scenario is resolved in one of two ways: either the slow consumer revives again and consumes all of the messages from the temporary file; or the broker determines that the slow consumer has died and the temporary file can be discarded.

Discarding messages

By default, when flow control is disabled and the relevant memory limit is reached, the slow consumer's messages are backed up in a temporary file. An alternative strategy for coping with the excess messages, however, is simply to discard the slow consumer's messages when they exceed a certain limit (where the oldest messages are discarded first). This strategy avoids the overhead of writing to a temporary file.

For example, if the slow consumer is receiving a feed of real-time stock quotes, it might be acceptable to discard older, undelivered stock quotes, because the information becomes stale.

To enable discarding of messages, define a *pending message limit strategy* in the broker configuration. For example, to specify that the backlog of messages stored in the broker (not including the prefetched messages) cannot exceed 10 for any topics that match the **PRICES.>** pattern (that is, topic names prefixed by **PRICES.**), configure the broker as follows:

```
<beans ... >
  <broker ...>
    <!--  lets define the dispatch policy -->
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic="PRICES.>">

            <!-- lets force old messages to be discarded for slow consumers -->
            <pendingMessageLimitStrategy>
              <constantPendingMessageLimitStrategy limit="10"/>
            </pendingMessageLimitStrategy>

          </policyEntry>
          ...
        </policyEntries>
      </policyMap>
    </destinationPolicy>
  </broker>
</beans>
```

For more details about how to configure pending message limit strategies, see http://activemq.apache.org/slow-consumer-handling.html.

How to turn off flow control

Flow control can be turned off by setting a destination policy in the broker's configuration. In particular, flow control can be enabled or disabled on individual destinations or groups of destinations (using wildcards). To disable flow control, set the **producerFlowControl** attribute to **false** on a **policyEntry** element.

For example, to configure a broker to disable flow control for all topic destinations starting with **FOO.**, insert a policy entry like the following into the broker's configuration:

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic="FOO.>" producerFlowControl="false"/>
        ...
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

Defining the memory limits

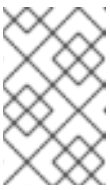
When flow control is enabled, the point at which flow control activates depends on the defined memory limits, which can be specified at either of the following levels of granularity:

- *Per-broker*—to set global memory limits on a broker, define a **systemUsage** element as a child of the **broker** element, as follows:

```
<broker>
 ...
 <systemUsage>
  <systemUsage>
   <memoryUsage>
    <memoryUsage limit="64 mb" />
   </memoryUsage>
   <storeUsage>
    <storeUsage limit="100 gb" />
   </storeUsage>
   <tempUsage>
    <tempUsage limit="10 gb" />
   </tempUsage>
  </systemUsage>
 </systemUsage>
 ...
</broker>
```

  Where the preceding sample specifies three distinct memory limits, as follows:

  - **memoryUsage**—specifies the maximum amount of memory allocated to the broker.
  - **storeUsage**—for persistent messages, specifies the maximum disk storage for the messages.



NOTE

In certain scenarios, the actual disk storage used by JBoss A-MQ can exceed the specified limit. For this reason, it is recommended that you set **storeUsage** to about 70% of the intended maximum disk storage.

  - **tempUsage**—for temporary messages, specifies the maximum amount of memory.

  The values shown in the preceding example are the defaults.

- *Per-destination*—to set a memory limit on a destination, set the **memoryLimit** attribute on the **policyEntry** element. The value of **memoryLimit** can be a string, such as **10 MB** or **512 KB**. For example, to limit the amount of memory on the **FOO.BAR** queue to 10 MB, define a policy entry like the following:

```
<policyEntry queue="FOO.BAR" memoryLimit="10 MB"/>
```

Making a producer aware of flow control

When a producer is subject to flow control, the default behaviour is for the **send()** operation to block, until enough memory is freed up in the broker for the producer to resume sending messages. If you want

the producer to be made aware of the fact that the **send()** operation is blocked due to flow control, you can enable either of the following attributes on the **systemUsage** element:

sendFailIfNoSpace

If **true**, the broker immediately returns an error when flow control is preventing producer **send()** operations; otherwise, revert to default behaviour.

sendFailIfNoSpaceAfterTimeout

Specifies a timeout in units of milliseconds. When flow control is preventing producer **send()** operations, the broker returns an error, after the specified timeout has elapsed.

The following example shows how to configure the broker to return an error to the producer immediately, whenever flow control is blocking the producer **send()** operations:

```
<broker>
  ...
  <systemUsage>
    <systemUsage sendFailIfNoSpace="true">
      <memoryUsage>
        <memoryUsage limit="64 mb" />
      </memoryUsage>
      ...
    </systemUsage>
  </systemUsage>
  ...
</broker>
```

CHAPTER 5. MANAGING SLOW CONSUMERS

OVERVIEW

Slow consumers are consumers whose dispatch buffer is regularly too full; the broker cannot dispatch messages to them because they have reached the prefect limit. This can bog down message processing in a number of ways and it can mask problems with a client. One of the major ways it can bog down a broker is by increasing its memory foot print by forcing the broker to hold a large number of messages in memory.

JBoss A-MQ provides two ways of limiting the impact of slow consumers:

- limiting the number of messages retained for a consumer

  When using non-durable topics, you can specify the number of messages that a destination will hold for a consumer. Once the limit is reached, older messages are discarded when new messages arrive.

- aborting slow consumers

  JBoss A-MQ determines slowness by monitoring how often a consumer's dispatch buffer is full. You can specify that consistently slow consumers be aborted by closing its connection to the broker.

LIMITING MESSAGE RETENTION

IMPORTANT

This strategy only works for topics. For queues, the way to manage the number of pending messages is through the message expiry setting.

Topics typically retain a copy of all unacknowledged messages for each of the consumers subscribed to it. For non-durable topics, the messages are stored in the broker's volatile memory, so if messages begin to pile up the broker's memory footprint begins to balloon.

To address this issue you can set the pending message limit strategy (**pendingMessageLimitStrategy**) on a topic to control the number of messages that are held for slow consumers. When set, the topic will retain the specified number of messages in addition to the consumer's prefetch limit.

IMPORTANT

The default setting for the strategy is **1000**, which means that the topic retains the newest 1000 unconsumed messages for a consumer. .

There are two ways to configure the pending message limit strategy:

- specifying a constant number of messages over the prefetch limit

  The **constantPendingMessageLimitStrategy** implementation allows you to specify constant number of messages to retain as shown in Example 5.1, “Constant Pending Message Limiter” .

  Example 5.1. Constant Pending Message Limiter

```
<broker ... >
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic=">" >
          <pendingMessageLimitStrategy>
            <constantPendingMessageLimitStrategy limit="50"/>
          </pendingMessageLimitStrategy>
        </policyEntry>
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

- specifying a multiplier that is applied to the prefetch limit

  The **prefetchRatePendingMessageLimitStrategy** implementation allows you to specify a multiplier that is applied to the prefect limit. Example 5.2, “Prefetch Limit Based Pending Message Limiter” shown configuration that retains twice the prefect limit. So if the prefect limit is 3, the destination will retain 6 pending messages for each consumer.

Example 5.2. Prefetch Limit Based Pending Message Limiter

```
<broker ... >
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic=">" >
          <pendingMessageLimitStrategy>
            <prefetchRatePendingMessageLimitStrategy multiplier="2"/>
          </pendingMessageLimitStrategy>
        </policyEntry>
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

HANDLING ADVISORY TOPICS FOR A BROKER NETWORK

A broker network relies on advisory messages to function properly so that it can propagate the demand for messages for a particular queue. Typically, if you run a network of brokers, you should not restrict the number of messages on advisory topics.

If you configure a <constantPendingMessageLimitStrategy> implementation for advisory topics, you run the risk of advisory messages getting discarded by the receiving broker and as a result demand subscriptions might not get created.

Instead, explicitly configure a policy entry for advisory topics so that messages on these topics are not discarded. Specify a configuration similar to the following example:

Example 5.3. Configuration for advisory topics

```
<policyEntry topic=">" producerFlowControl="true" memoryLimit="10mb">
  <pendingMessageLimitStrategy>
    <constantPendingMessageLimitStrategy limit="1000" />
  </pendingMessageLimitStrategy>
</policyEntry>
...
<policyEntry topic="ActiveMQ.Advisory.>" producerFlowControl="true" memoryLimit="10mb" />
```

In this example, the broker does not discard advisory topics. For any other (non-advisory) topics, the broker retains only the newest 1000 messages. You can change the **constantPendingMessageLimitStrategy** limit value based on your application requirements.

For more information about **constantPendingMessageLimitStrategy**, go to http://activemq.apache.org/slow-consumer-handling.html

ABORTING SLOW CONSUMERS

Another strategy for managing slow consumers is to have the broker detect slow consumers and automatically abort consumers that are consistently slow. When a slow consumer is aborted, its connection to the broker is closed.

The broker marks a consumer slow when the broker has messages to dispatch to the consumer, but the consumer's prefect buffer is full. As the consumer acknowledges consumption of messages from the prefect buffer and the broker can once again start dispatching messages to the consumer, the broker will stop considering the consumer slow. If the consumer's prefect buffer fills up again, the broker will again mark the consumer as slow.

The abort slow consumers strategy allows the broker to abort consumers when one of two conditions is met:

- a consumer is considered slow for specified amount of time
- a consumer is considered slow a specified number of times

The abort slow consumer strategy is activated by adding the configuration shown in Example 5.4, “Aborting Slow Consumers” to a destination's configuration.

Example 5.4. Aborting Slow Consumers

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic=">" >
          <slowConsumerStrategy>
            <abortSlowConsumerStrategy />
          </slowConsumerStrategy>
        </policyEntry>
      </policyEntries>
    </policyMap>
```

```
    </destinationPolicy>
    ...
</broker>
```

The **abortSlowConsumerStrategy** element activates the abort slow consumer strategy with default settings. Consumers that are considered slow for more than 30 seconds are aborted. You can modify when slow consumers are aborted using the attributes described in Table 5.1, "Settings for Abort Slow Consumer Strategy".

Table 5.1. Settings for Abort Slow Consumer Strategy

Attribute	Default	Description
maxSlowCount	**-1**	Specifies the number of times a consumer can be considered slow before it is aborted. **-1** specifies that a consumer can be considered slow an infinite number of times.
maxSlowDuration	**30000**	Specifies the maximum amount of time, in milliseconds, that a consumer can be continuously slow before it is aborted.
checkPeriod	**30000**	Specifies, in milliseconds, the time between checks for slow consumers.
abortConnection	**false**	Specifies whether the broker forces the consumer connection to close. The default value specifies that the broker will send a message to the consumer requesting it to close its connection. **true** specifies that the broker will automatically close the consumer's connection.

For example, Example 5.5, "Aborting Repeatedly Slow Consumers" shows configuration for aborting consumers that have been marked as slow 30 times.

Example 5.5. Aborting Repeatedly Slow Consumers

```
<abortSlowConsumerStrategy maxSlowCount="30" />
```

CHAPTER 6. PERSISTENT MESSAGING

Abstract

This chapter outlines the tuning techniques that can be used to optimize the performance of a persistent broker. Hence, the tuning techniques in this chapter are focused mainly on the interaction between the broker and its message store.

6.1. SERIALIZING TO DISK

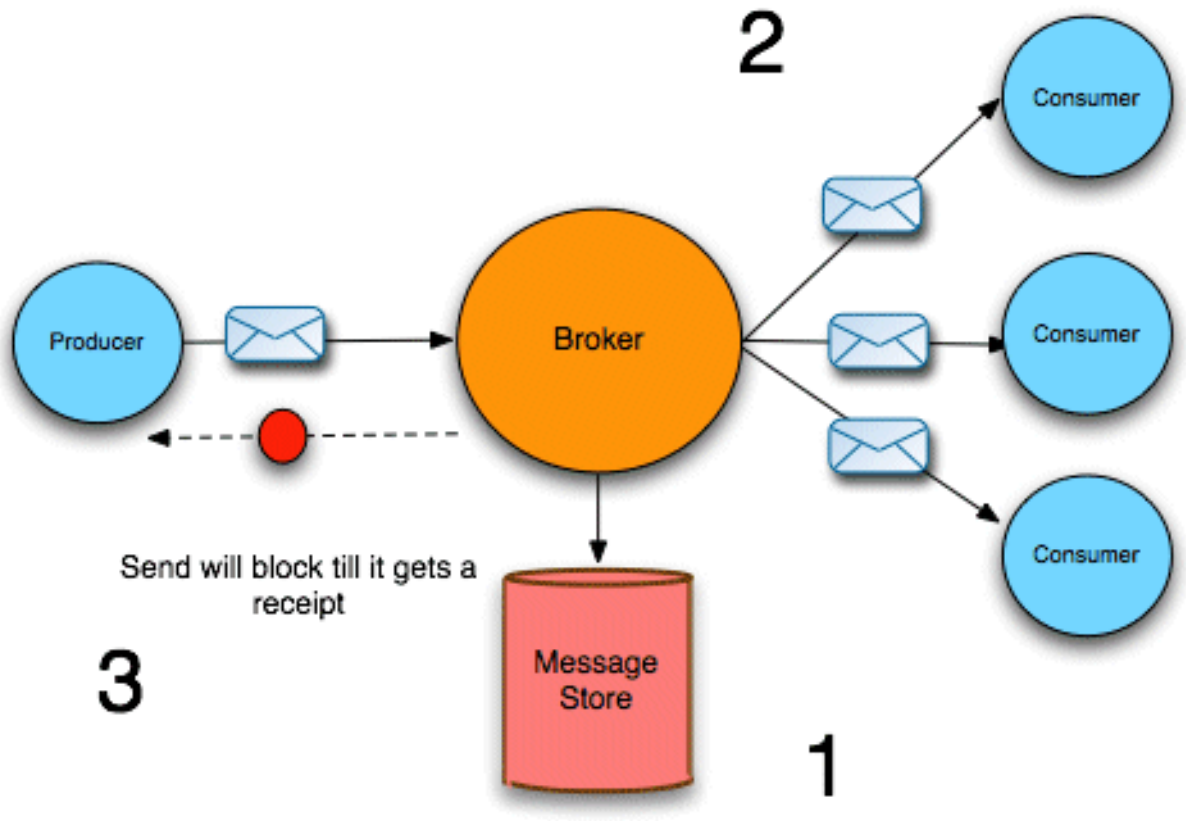
KahaDB message store

KahaDB is the recommended message store to use with Red Hat JBoss A-MQ in order to achieve maximum performance. The KahaDB supports several options that you can customize to obtain optimum performance.

Synchronous dispatch through a persistent broker

Figure 6.1, "Synchronous Dispatch through a Persistent Broker" gives an overview of the sequence of steps for a message dispatched synchronously through a persistent broker.

Figure 6.1. Synchronous Dispatch through a Persistent Broker



After receiving a message from a producer, the broker dispatches the messages to the consumers, as follows:

1. The broker pushes the message into the message store. Assuming that the **enableJournalDiskSyncs** option is **true**, the message store also writes the message to disk, before the broker proceeds.

2. The broker now sends the message to all of the interested consumers (but *does not wait for consumer acknowledgements*). For topics, the broker dispatches the message immediately, while for queues, the broker adds the message to a destination cursor.

3. The broker then sends a receipt back to the producer. The receipt can thus be sent back *before* the consumers have finished acknowledging messages (in the case of topic messages, consumer acknowledgements are usually not required anyway).

Concurrent store and dispatch

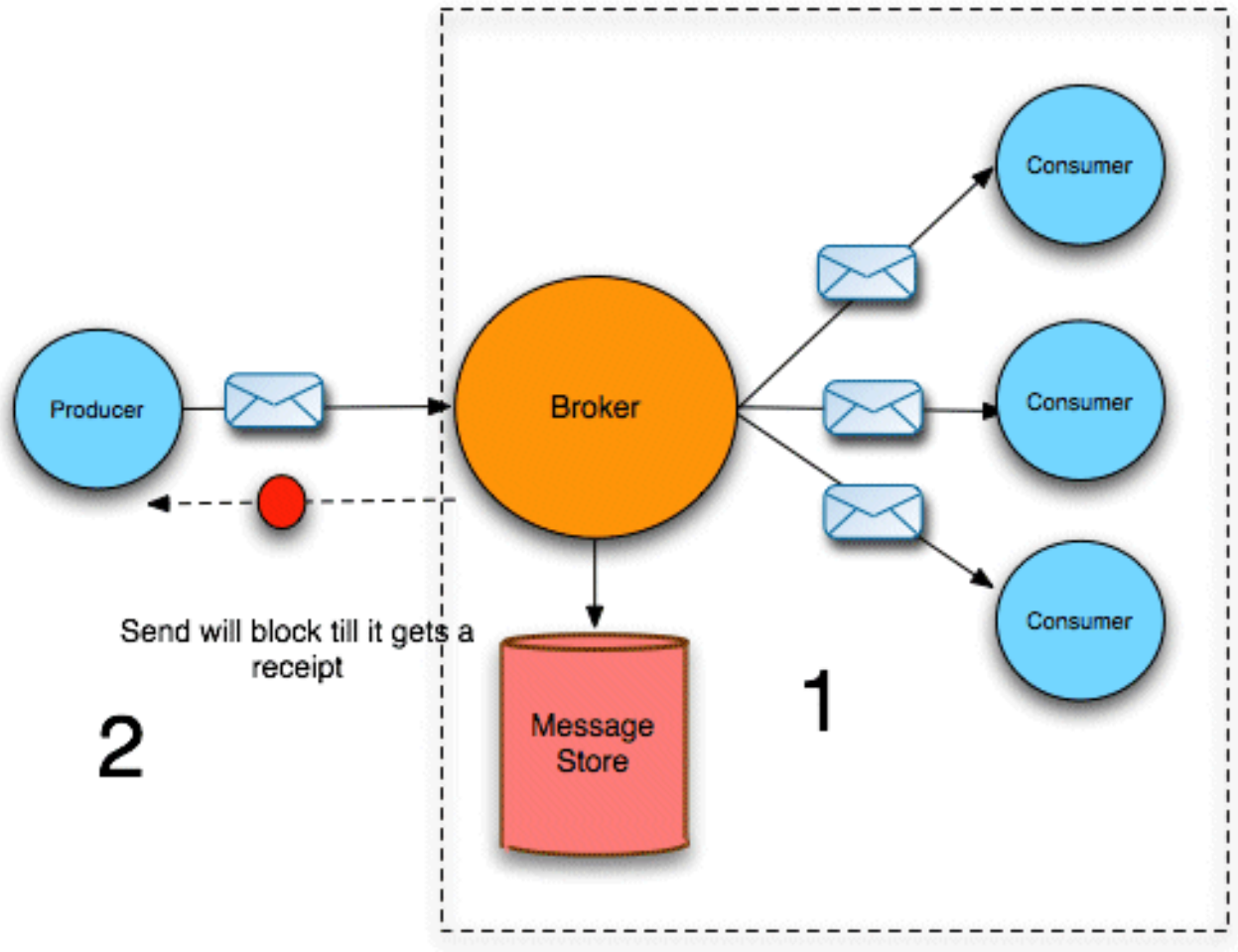
To speed up the performance of the broker, you can enable the *concurrent store and dispatch* optimization, which allows storing the message and sending to the consumer to proceed concurrently.

NOTE

Concurrent store and dispatch is enabled, by default, for queues.

Figure 6.1, "Synchronous Dispatch through a Persistent Broker" gives an overview of message dispatch when the concurrent store and dispatch optimization is enabled.

Figure 6.2. Concurrent Store and Dispatch



After receiving a message from a producer, the broker dispatches the messages to the consumers, as follows:

1. The broker pushes the message onto the message store and, *concurrently*, sends the message to all of the interested consumers. After sending the message to the consumers, the broker then sends a receipt back to the producer, without waiting for consumer acknowledgements or for the message store to synchronize to disk.

2. As soon as the broker receives acknowledgements from all the consumers, the broker removes the message from the message store. Because consumers typically acknowledge messages faster than a message store can write them to disk, this often means that write to disk is optimized away entirely. That is, the message is removed from the message store before it is ever physically written to disk.

One drawback of concurrent store and dispatch is that it does reduce reliability.

Configuring concurrent store and dispatch

The concurrent store and dispatch feature can be enabled separately for queues and topics, by setting the **concurrentStoreAndDispatchQueues** flag and the **concurrentStoreAndDispatchTopics** flag. By default, it is enabled for queues, but disabled for topics. To enable concurrent store and dispatch for *both* queues and topics, configure the **kahaDB** element in the broker configuration as follows:

```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
  ...
  <persistenceAdapter>
    <kahaDB directory="activemq-data"
            journalMaxFileLength="32mb"
            concurrentStoreAndDispatchQueues="true"
            concurrentStoreAndDispatchTopics="true"
            />
  </persistenceAdapter>
</broker>
```

Reducing memory footprint of pending messages

After a queue message is written to persistent storage, a copy of the message remains in memory, pending dispatch to a consumer. If the relevant consumer is very slow, however, this can lead to a build-up of messages in the broker and, in some cases, can lead to an out-of-memory error. If you observer this problem in your broker, you can enable an option to reduce the memory footprint of the pending messages; but you should note that this option is *not compatible with concurrent store and dispatch* .

To reduce the memory footprint of pending queue messages, define a destination policy for the relevant queues, enabling the **reduceMemoryFootprint** option, as follows:

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry queue=">" reduceMemoryFootprint="true" />
      </policyEntries>
    </policyMap>
```

```
    </destinationPolicy>
    ...
</broker>
```

When the **reduceMemoryFootprint** option is enabled, a message's marshalled content is cleared immediately after the message is written to persistent storage. This results in approximately a 50% reduction in the amount of memory occupied by the pending messages.

6.2. KAHADB OPTIMIZATION

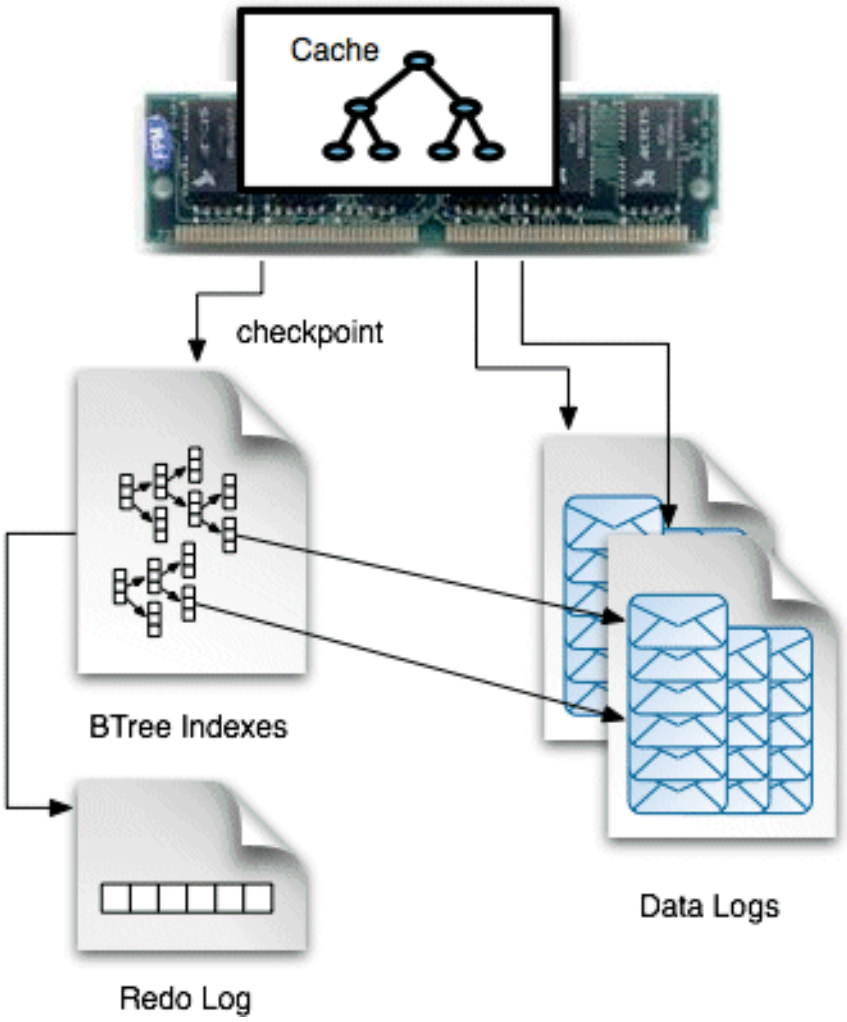
Overview

The Red Hat JBoss A-MQ message store has undergone a process of evolution. Currently, the KahaDB message store is the default (and recommended) message store, while the AMQ message store and the (original) kaha message store represent earlier generations of message store technology.

KahaDB architecture

The KahaDB architecture—as shown in Figure 6.3, "KahaDB Architecture"—is designed to facilitate high-speed message storage and retrieval. The bulk of the data is stored in rolling journal files (*data logs*), where all broker events are continuously appended. In particular, pending messages are also stored in the data logs.

Figure 6.3. KahaDB Architecture



In order to facilitate rapid retrieval of messages from the data logs, a *B-tree index* is created, which contains pointers to the locations of all the messages embedded in the data log files. The complete B-tree index is stored on disk and part or all of the B-tree index is held in a cache in memory. Evidently, the B-tree index can work more efficiently, if the complete index fits into the cache.

Sample configuration

The following example shows how to configure a broker to use the KahaDB message store, by adding a **persistenceAdapter** element containing a **kahaDB** child element:

```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
  ...
  <persistenceAdapter>
 <kahaDB directory="activemq-data" journalMaxFileLength="32mb"/>
  </persistenceAdapter>
</broker>
```

The **directory** property specifies the directory where the KahaDB files are stored and the **journalMaxFileLength** specifies the maximum size of a data log file.

Performance optimization

You can optimize the performance of the KahaDB message store by modifying the following properties (set as attributes on the **kahaDB** element):

- **indexCacheSize**—(default **10000**) specifies the size of the cache in units of pages (where one page is 4 KB by default). Generally, the cache should be as large as possible, to avoid swapping pages in and out of memory. Check the size of your metadata store file, **db.data**, to get some idea of how big the cache needs to be.

- **indexWriteBatchSize**—(default **1000**) defines the threshold for the number of dirty indexes that are allowed to accumulate, before KahaDB writes the cache to the store. If you want to maximize the speed of the broker, you could set this property to a large value, so that the store is updated *only* during checkpoints. But this carries the risk of losing a large amount of metadata, in the event of a system failure (causing the broker to restart very slowly).

- **journalMaxFileLength**—(default **32mb**) when the throughput of a broker is very large, you can fill up a journal file quite quickly. Because there is a cost associated with closing a full journal file and opening a new journal file, you can get a slight performance improvement by increasing the journal file size, so that this cost is incurred less frequently.

- **enableJournalDiskSyncs**—(default **true**) normally, the broker performs a disk sync (ensuring that a message has been physically written to disk) before sending the acknowledgement back to a producer. You can obtain a substantial improvement in broker performance by disabling disk syncs (setting this property to **false**), but this reduces the reliability of the broker somewhat.

WARNING

If you need to satisfy the JMS durability requirement and be certain that you do not lose any messages, do *not* disable journal disk syncs.

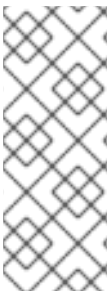
For more details about these KahaDB configuration properties, see *Configuring Broker Persistence*.

Optimizing disk syncs

On some operating systems, you can obtain better performance by configuring KahaDB to call the **fdatasync()** system call instead of the **fsync()** system call (the default), when writing to a file. The difference between these system calls is that **fdatasync()** updates only the file data, whereas **fsync()** updates both the file data *and* the file metadata (for example, the access time).

To enable this optimization, add the following system property setting to the **etc/system.properties** file in your JBoss A-MQ installation:

```
org.apache.activemq.kahaDB.files.skipMetadataUpdate=true
```



NOTE

This optimization might not be effective on all operating systems, because it ultimately relies on the Java Virtual Machine (JVM) implementation to make the **fdatasync()** system call. When this option is enabled, the JBoss A-MQ runtime actually makes a call to **java.nio.channels.FileChannel#force=false**. For some JVMs, this can result in a call to **fdatasync()** (so that the optimization is effective), but with other JVMs it might be implemented using **fsync()** (so that the optimization has no effect).

NOTE

For users of Red Hat Enterprise Linux (RHEL), the implementation of **fsync()** on RHEL 6 is noticeably slower than on RHEL 4 (this is due to a bug fix in RHEL 6). So, this optimization works particularly well on the RHEL 6 platform, where **fdatasync()** is significantly faster.

6.3. VMCURSOR ON DESTINATION

Overview

In Red Hat JBoss A-MQ, a cursor is used to hold a batch of messages in memory, while those messages are waiting to be sent to a destination. By default, the batch of messages is pulled out of the message store and then held in the cursor (this is the *store cursor*).

JBoss A-MQ has another cursor implementation, the *VM cursor*, which can be significantly faster in some cases. With the VM cursor, incoming messages are inserted *directly* into the cursor, bypassing the message store (the messages are also, concurrently, inserted into the message store). This works well if the consumers are fast and are able to keep up with the flow of messages. On the other hand, for slow consumers this strategy does not work so well, because the VM cursor fills up with a backlog of messages and then it becomes necessary to invoke flow control to throttle messages from the producer.

Configuring destinations to use the vmCursor

To configure a broker to use the **vmCursor** for all topics *and* queues, add the following lines to your broker configuration:

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic=">">
          <pendingSubscriberPolicy>
            <vmCursor />
          </pendingSubscriberPolicy>
        </policyEntry>
        <policyEntry queue=">">
          <deadLetterStrategy>
            <individualDeadLetterStrategy queuePrefix="Test.DLQ."/>
          </deadLetterStrategy>
          <pendingQueuePolicy>
            <vmQueueCursor />
          </pendingQueuePolicy>
        </policyEntry>
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

Where both the topic and queue entries specify the wildcard, **>**, that matches all destination names. You could also specify a more selective destination pattern, so that the VM cursor would be enabled only for those destinations where you are sure that consumers can keep up with the message flow.

Reference

For more information about the Red Hat JBoss A-MQ cursor implementations–, and the advantages and disadvantages of each one see *Configuring Broker Persistence*.

6.4. JMS TRANSACTIONS

Improving efficiency using JMS transactions

You can improve efficiency of the broker using JMS transactions, because JMS transactions enable the broker to process messages in *batches*. That is, a batch consists of all the messages a producer sends to the broker before calling commit. Sending messages in batches improves the performance of the persistence layer, because the message store is not required to write the batched messages to disk until commit is called. Hence, the message store accesses the file system less frequently—that is, once per transaction instead of once per message.