



# Fuse Message Broker

## Tuning the Broker

Version 5.5  
February 2012



# Tuning the Broker

Version 5.5

Updated: 27 Mar 2014

Copyright © 2012-2013 Red Hat, Inc. and/or its affiliates.

## **Trademark Disclaimer**

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Apache, ServiceMix, Camel, CXF, and ActiveMQ are trademarks of Apache Software Foundation. Any other names contained herein may be trademarks of their respective owners.

## **Third Party Acknowledgements**

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (<http://jline.sourceforge.net>) jline:jline:jar:1.0

License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux <mwp1@cornell.edu>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (<http://woodstox.codehaus.org/StAX2>) org.codehaus.woodstox:stax2-api:jar:3.1.1

License: The BSD License (<http://www.opensource.org/licenses/bsd-license.php>)

Copyright (c) <YEAR>, <OWNER> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (<http://www.jibx.org/main-reactor/jibx-run>) org.jibx:jibx-run:bundle:1.2.3

License: BSD (<http://jibx.sourceforge.net/jibx-license.html>) Copyright (c) 2003-2010, Dennis M. Sosnoski.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (<http://www.jboss.org/javassist>) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile  
License: MPL (<http://www.mozilla.org/MPL/MPL-1.1.html>)
- HAPI-OSGI-Base Module (<http://hl7api.sourceforge.net/hapi-osgi-base/>) ca.uhn.hapi:hapi-osgi-base:bundle:1.2  
License: Mozilla Public License 1.1 (<http://www.mozilla.org/MPL/MPL-1.1.txt>)



# Table of Contents

<b>1. General Tuning Techniques .....</b>	<b>11</b>
Introduction to Performance Tuning .....	12
System Environment .....	13
Co-locating the Broker .....	14
Optimizing the Protocols .....	16
Message Encoding .....	19
Consumer Performance .....	20
Acknowledgment Modes .....	21
Reducing Context Switching .....	23
Prefetch Limit .....	27
Producer Performance .....	31
Async Sends .....	32
Flow Control .....	34
Threading Optimizations .....	40
Vertical Scaling .....	41
Horizontal Scaling .....	42
Integration with Spring and Camel .....	45
<b>2. Persistent Messaging .....</b>	<b>47</b>
Serializing to Disk .....	48
KahaDB Optimization .....	52
vmCursor on Destination .....	55
JMS Transactions .....	57

# List of Figures

1.1. Broker Co-located with Producer .....	14
1.2. Default Consumer Threading Model .....	24
1.3. Optimized Consumer Threading Model .....	25
1.4. Consumer Prefetch Limit .....	27
1.5. Broker with Flow Control Enabled .....	34
1.6. Broker with Flow Control Disabled .....	35
1.7. Scaling with Multiple Brokers .....	42
2.1. Synchronous Dispatch through a Persistent Broker .....	48
2.2. Concurrent Store and Dispatch .....	49
2.3. KahaDB Architecture .....	52



# List of Tables

1.1. OpenWire Parameters Affecting Performance ..... 17



# Chapter 1. General Tuning Techniques

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

Introduction to Performance Tuning .....	12
System Environment .....	13
Co-locating the Broker .....	14
Optimizing the Protocols .....	16
Message Encoding .....	19
Consumer Performance .....	20
Acknowledgment Modes .....	21
Reducing Context Switching .....	23
Prefetch Limit .....	27
Producer Performance .....	31
Async Sends .....	32
Flow Control .....	34
Threading Optimizations .....	40
Vertical Scaling .....	41
Horizontal Scaling .....	42
Integration with Spring and Camel .....	45

# 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 the next chapter, ["Persistent Messaging" on page 47](#).

## 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 ["Horizontal Scaling" on page 42](#), but for a comprehensive discussion and explanation of how to set up a broker network, please consult the [Using Networks of Brokers](#).

# System Environment

## Overview

Before discussing how to tune the performance of an Fuse Message Broker 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 Fuse Message Broker tuning, if the underlying network is very slow. One strategy that you can try is to enable compression of large messages (see ["Enabling compression" on page 18](#)). 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 behavior.

## 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.

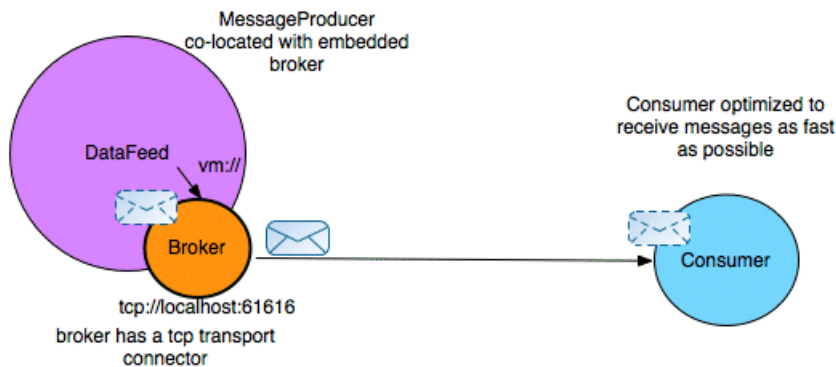
# Co-locating the Broker

## 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 1.1 on page 14 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 Fuse Message Broker's `vm://` transport.

**Figure 1.1. 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 Fuse Message Broker). 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, a 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 ["VM Transport"](#) in *Connectivity Guide*.

## 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 behavior, the calling thread can actually dispatch directly to consumers.

# Optimizing the Protocols

## Overview

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

- "TCP transport" on page 16.
- "OpenWire protocol" on page 16.
- "Enabling compression" on page 18.

## 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:

$$\text{Buffer Size} = \text{Bandwidth} \times \text{Round-Trip-Time}$$

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](http://en.wikipedia.org/wiki/Network_Improvement)<sup>1</sup>.

- *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 1.1 on page 17](#).

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Network\\_Improvement](http://en.wikipedia.org/wiki/Network_Improvement)



**Table 1.1. 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 <a href="http://en.wikipedia.org/wiki/Nagles_algorithm">Nagles algorithm</a> <sup>2</sup> 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
```

<sup>2</sup> [http://en.wikipedia.org/wiki/Nagles\\_algorithm](http://en.wikipedia.org/wiki/Nagles_algorithm)

## 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
```

# 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](http://code.google.com/p/protobuf/)<sup>3</sup>, which has excellent performance characteristics.

---

<sup>3</sup> <http://code.google.com/p/protobuf/>

# Consumer Performance

Acknowledgment Modes .....	21
Reducing Context Switching .....	23
Prefetch Limit .....	27

# Acknowledgment Modes

## Overview

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

## Supported acknowledgment modes

Fuse Message Broker supports the following acknowledgment 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 acknowledgment 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 acknowledgment.

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 acknowledgments 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 acknowledgment 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
```

## Choosing the acknowledgment 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 Fuse Message Broker in combination with Apache Camel, you can easily use the [Idempotent Consumer](http://camel.apache.org/idempotent-consumer.html)<sup>4</sup> pattern to implement duplicate detection.

---

<sup>4</sup> <http://camel.apache.org/idempotent-consumer.html>

# Reducing Context Switching

## Overview

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

- ["Optimize message dispatching on the broker side" on page 23.](#)
- ["Optimize message reception on the consumer side" on page 23.](#)

## 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);
```

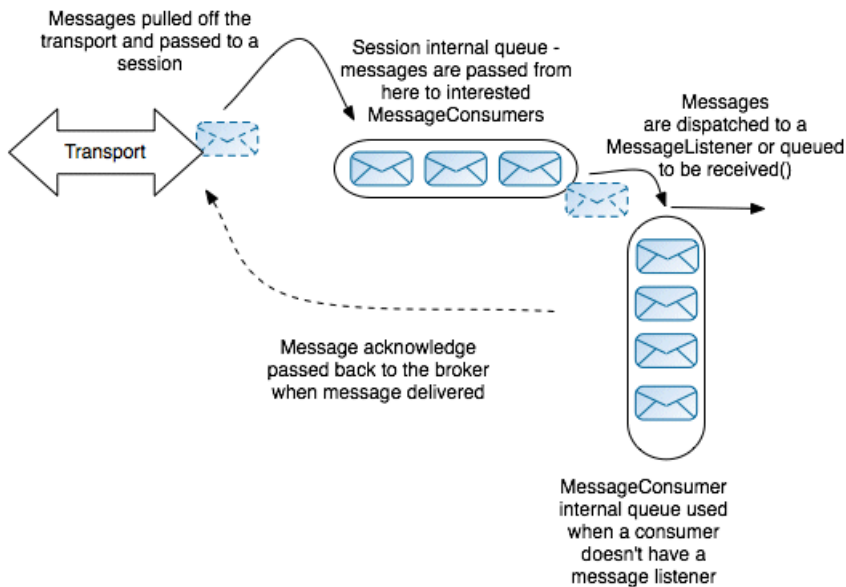
## Optimize message reception on the consumer side

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 1.2 on page 24 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.

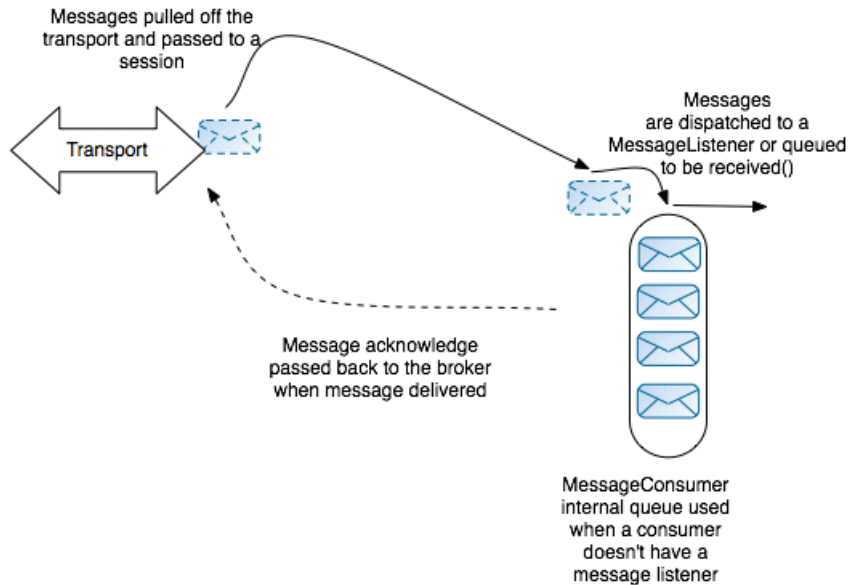
**Figure 1.2. Default Consumer Threading Model**



## Optimized consumer threading model

Figure 1.3 on page 25 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 1.3. 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 acknowledgment 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`).

## 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();

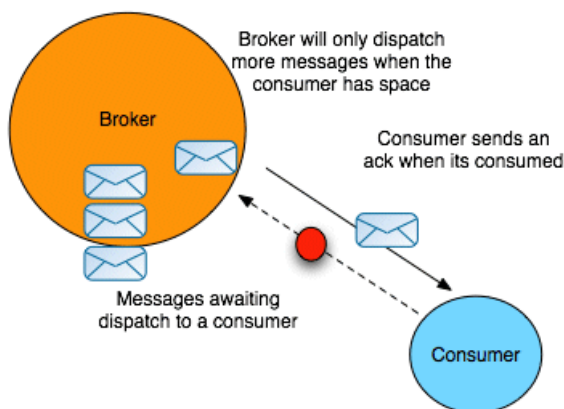
// Create the one-and-only session on this connection.
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

## Prefetch Limit

### Overview

Figure 1.4 on page 27 illustrates the behavior of a broker, as it waits to receive acknowledgments for the messages it has already sent to a consumer.

**Figure 1.4. Consumer Prefetch Limit**



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 acknowledgments.

### Default prefetch limits

Different prefetch limits can be set for each consumer type. The default prefetch limits are as follows:

#### 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.

## How to set prefetch limits

You can set prefetch limits either on the broker or on the consumer and there are three levels of granularity to choose from, as follows:

- "Per broker" on page 28.
- "Per connection factory" on page 29.
- "Per destination" on page 30.

### Per broker

You can define the prefetch limits for all consumers that attach to a particular broker, by setting a destination policy on the broker. To set the destination policy, add a `destinationPolicy` element as a child of the `broker` element in the broker's configuration, as follows:

```
<broker ... >
...
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry queue="queue.>" queuePrefetch="1"/>
      <policyEntry topic="topic.>" topicPrefetch="1000"/>
    </policyEntries>
  </policyMap>
</destinationPolicy>
...
</broker>
```

In the preceding example, the queue prefetch limit for all queues whose names start with `queue.` is set to 1 (the `>` character is a wildcard symbol that matches one or more name segments); and the topic prefetch limit for all topics whose names start with `topic.` is set to 1000.

The following attributes of `policyEntry` are used to specify prefetch limits:

`queuePrefetch`

Specifies the queue consumer prefetch limit.

`queueBrowserPrefetch`

Specifies the queue browser prefetch limit.

`topicPrefetch`

Specifies the topic consumer prefetch limit.

`durableTopicPrefetch`

Specifies the durable topic subscriber prefetch limit.

## Per connection factory

In a consumer, you can specify the prefetch limits on a connection factory by setting properties on an `ActiveMQConnectionFactory` instance. For example, the following code example shows how to specify the prefetch limits for all consumer types on a connection factory:

```
// Java
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();

Properties props = new Properties();
props.setProperty("prefetchPolicy.queuePrefetch", "1000");
props.setProperty("prefetchPolicy.queueBrowserPrefetch", "500");
props.setProperty("prefetchPolicy.durableTopicPrefetch", "100");
props.setProperty("prefetchPolicy.topicPrefetch", "32766");
```

```
factory.setProperties(props);
```

## Per destination

At the finest level of granularity, you can specify the prefetch limit on each destination instance that you create in a consumer. For example, to consume from the queue, `TEST.QUEUE`, with a prefetch limit of 10, create a `MessageConsumer` instance as follows:

```
// Java
Queue queue =
    new ActiveMQQueue("TEST.QUEUE?consumer.prefetchSize=10");
MessageConsumer consumer = session.createConsumer(queue);
```

# Producer Performance

Async Sends ..... 32

Flow Control ..... 34

## 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.

Fuse Message Broker 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 acknowledgment 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://localhost: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);
```

## Flow Control

### Overview

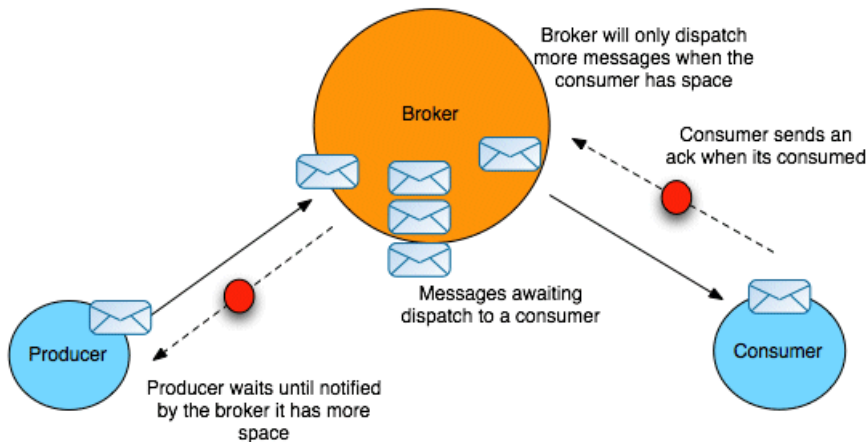
Memory limits, when configured, prevent the broker from running out of resources. The default behavior, when a limit is reached, is to block the sending thread in the broker, which blocks the destination and the connection.

Producer flow control is a mechanism that pushes the blocking behavior 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 1.5 on page 34 gives an overview of what happens to a messaging application when flow control is enabled.

**Figure 1.5. 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.

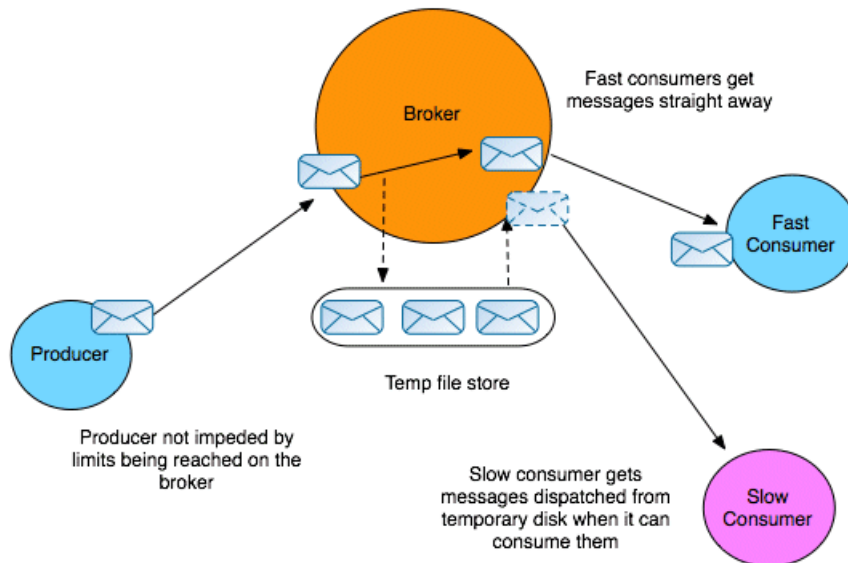
### Note

There are some differences in behavior 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 ["vmCursor on Destination" on page 55](#) 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 1.6 on page 35](#).

**Figure 1.6. 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 `F00.`, insert a policy entry like the following into the broker's configuration:

```
<broker ... >
...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic="F00.>" 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>
```

```
</systemUsage>
...
</broker>
```

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

- `memoryUsage`—for non-persistent messages, specifies the maximum amount of memory used to hold the messages.
- `storeUsage`—for persistent messages, specifies the maximum disk storage for the messages.
- `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 `F00.BAR` queue to 10 MB, define a policy entry like the following:

```
<policyEntry queue="F00.BAR" memoryLimit="10 MB"/>
```

## Making a producer aware of flow control

When a producer is subject to flow control, the default behavior 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 behavior.

`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>
```

```
    </memoryUsage>
    ...
  </systemUsage>
</systemUsage>
...
</broker>
```

# 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.



# 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 Fuse Message Broker:

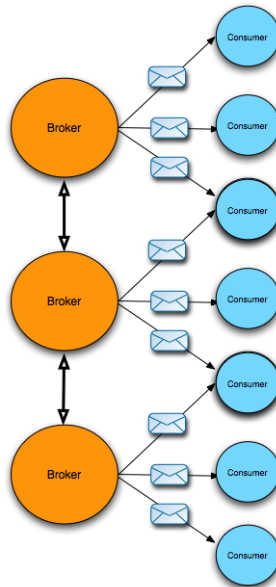
- *NIO transport on the broker*—to reduce the number of threads required, use the NIO transport (instead of the TCP transport) when defining transport connectors in the broker. *Do not use the NIO transport in clients*, it is only meant to be used in the broker.
- *Allocate more memory to broker*—to 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.

# Horizontal Scaling

## Overview

*Horizontal scaling* refers to the strategy of increasing capacity by adding multiple brokers to your messaging network. [Figure 1.7 on page 42](#) illustrates how a broker network can be used to support a large number of consumers.

**Figure 1.7. 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

Fuse Message Broker 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.

# 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 Apache ActiveMQ 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.

# Chapter 2. Persistent Messaging

*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.*

Serializing to Disk .....	48
KahaDB Optimization .....	52
vmCursor on Destination .....	55
JMS Transactions .....	57

## Serializing to Disk

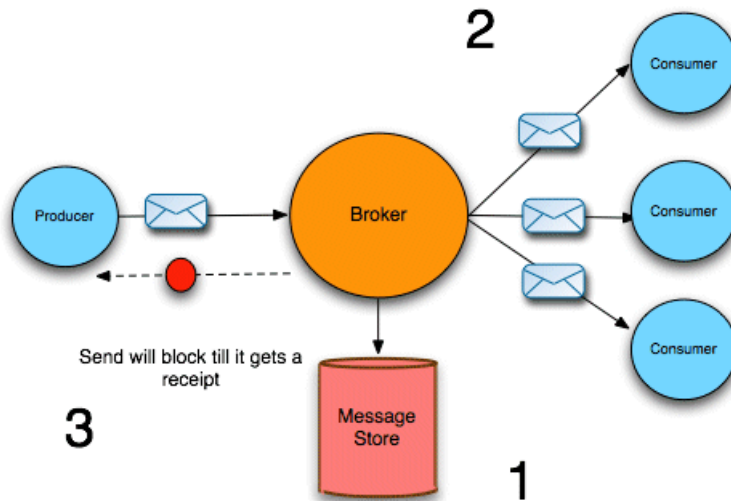
### KaHaDB message store

KaHaDB is the recommended message store to use with Fuse Message Broker 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 2.1 on page 48 gives an overview of the sequence of steps for a message dispatched synchronously through a persistent broker.

**Figure 2.1. 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 acknowledgments*). For topics, the broker dispatches the message immediately, while for queues, the broker adds the message to a destination cursor.



- 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 acknowledgments 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.

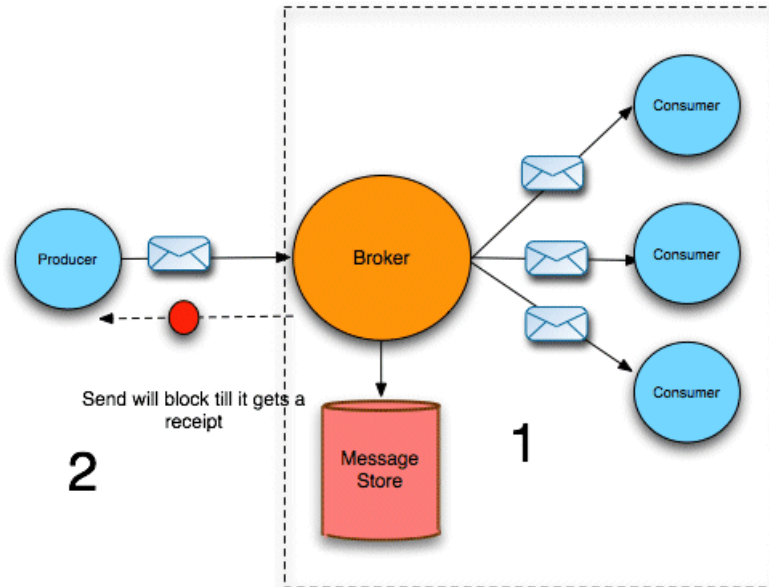


### Note

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

Figure 2.1 on page 48 gives an overview of message dispatch when the concurrent store and dispatch optimization is enabled.

**Figure 2.2. 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 acknowledgments or for the message store to synchronize to disk.
2. As soon as the broker receives acknowledgments 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>
```



### Note

In versions of Fuse Message Broker prior to 5.4.0, it was possible to enable concurrent store and dispatch for transactions as well, but this option is no longer supported. Transaction are now always executed serially (as in [Figure 2.1 on page 48](#)).

## 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 observe 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.

# KahaDB Optimization

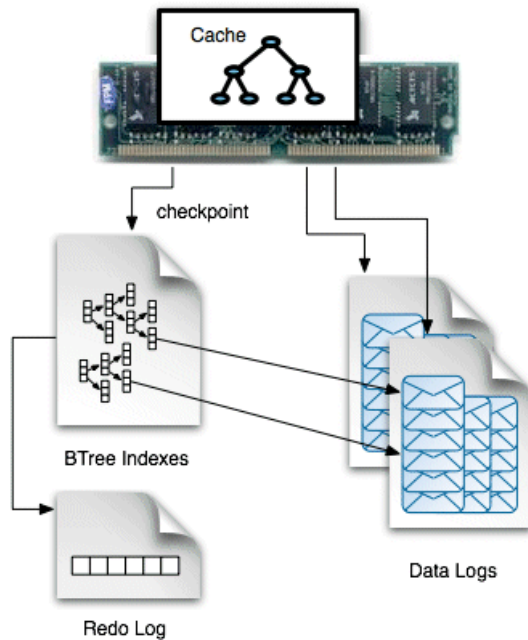
## Overview

The Fuse Message Broker 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 2.3 on page 52](#)—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 2.3. 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 acknowledgment 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 ["Optimizing the Metadata Cache"](#) in *Configuring Broker Persistence* and ["Configuring the KahaDB Message Store"](#) in *Configuring Broker Persistence*.

# vmCursor on Destination

## Overview

In Fuse Message Broker, 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*).

Fuse Message Broker 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="*>
          <pendingSubscriberPolicy>
            <vmCursor />
          </pendingSubscriberPolicy>
        </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 Fuse Message Broker cursor implementations—, and the advantages and disadvantages of each one—see ["Message Cursors"](#) in *Configuring Broker Persistence*.



# 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.

