High Performance Messaging

by Mark Richards

As Woody Allen once said "It is impossible to travel faster than the speed of light, and certainly not desirable, as one's hat keeps blowing off". While messages traveling through your system may never quite reach the speed of light, you could certainly make them travel fast. In this article I will explore four simple techniques that will increase the speed and throughput of your messaging system through relatively minor changes in your messaging infrastructure.

Introduction

Message-based systems represent some of the most flexible and reliable systems in the world. They also tend to be very fast due to their ability to process multiple messages concurrently. However, sometimes you find yourself in situations where your messaging infrastructure isn't quite fast enough to handle the volume of messages coming in. This is particularly true for high-volume systems like trading applications, telecommunication applications, or large e-commerce applications.

Fortunately, there are several simple techniques you can use to increase the overall performance and throughput of your messaging application. In this article I will describe four techniques you can use that are fairly simple to configure and implement. These techniques include tuning the configuration settings for competing consumers when using Spring Message-Driven POJOs, leveraging the pub/sub messaging model for point-to-point messaging, using multiple queues to increase performance and throughput, and tuning your messaging system with regards to message persistence.

Spring MDP Optimizations

With point-to-point messaging (queues) you can add multiple consumers to a queue, which allows you to process multiple messages concurrently. This integration pattern, known as *competing consumers*, is very effective in increasing the overall throughput of your messaging system, providing you are not concerned about the order in which messages are processed.

Using Spring Message-Driven POJOs (MDPs) you can very easily adjust the number of consumers by setting the concurrency property on the listener container located in your `app-context.xml` file, as illustrated in Listing RIC-1 below.

Here's where it gets interesting. When you set the concurrency property on the listener container to a specific number of consumers, Spring will create that many competing consumers, right? Wrong. Spring will leverage several heuristics based on wait times, memory, and other factors, and instantiate *up to* the number of concurrent consumers you specify. In reality the number of consumers Spring will create is far less than the number of consumers you specify. In other words, what you are really specifying in the concurrency property is the *maximum* number of concurrent (or competing) consumers. This behavior has profound impacts on performance and throughput.

For example, assume it takes 1 second to process a message. If you only have a single consumer listening on a queue and you send 60 messages to the queue at once, theoretically (ignoring overhead and instantiation times) it should take 60 seconds to process the messages. If you set the concurrency property in the preceding example to 60, meaning 60 concurrent consumers, it should theoretically take 1 second to process all 60 messages. However, using the preceding code, it takes a little over 33 seconds to process 60 messages—certainly not what you expected (or planned for).

```
<jms:listener-container connectionfactory="qcf" concurrency="60">
  <jms:listener destination="EQ.TRADE.Q" ref="messageListener"/>
</jms:listener-container>
```

Listing RIC-1

```
<jms:listener-container connectionfactory="qcf" concurrency="60-60">
  <jms:listener destination="EQ.TRADE.Q" ref="messageListener"/>
</jms:listener-container>
```

Listing RIC-2

To fix this problem, Spring offers a min-max option on the concurrency property. Setting the concurrency property, as shown in Listing RIC-2, using the min-max feature gives you exactly the number of concurrent consumers you specify (in this case 60).

Notice in the preceding code the simple change from the code before (*concurrency="60-60"*). Setting the concurrency value in this manner now gives you 60 concurrent consumers, regardless of wait times or memory. Running the same test as above, the elapsed time to process 60 messages at 1 second per message now drops from 33 seconds to around 3 seconds (the additional 2 seconds is for overhead). This is a significant performance improvement over the old setting and provides you with much better throughput.

Leveraging the Pub/Sub Model

The Java Message Service (JMS) offers two messaging models: point-to-point and publish-and-subscribe (pub/sub). The point-to-point model uses queues and is typically used to deliver messages to a specific consumer for processing. With point-to-point you are guaranteed that only one consumer will receive a message sent to a queue, regardless of the number of competing consumers you have listening on the queue (see the previous section).

The pub/sub model uses topics and is typically used for broadcasting messages. Unlike the point-to-point model, all subscribers (consumers) listening on a topic will receive the message. As a result, the pub/sub model is generally not used for processing business-related messages intended for consumption by only one consumer.

Although the point-to-point messaging model supports competing consumers and load balancing for high throughput, there are many situations in which competing consumers cannot be used. Most of these situations require the message order to be preserved during processing. For example, in financial trading systems you cannot process a trade cancellation prior to a trade order. Similarly, in banking you generally do not want to credit one account without first debiting the other. In systems such as these where the order in which messages are processed matters, you generally cannot take advantage of competing consumers. You are therefore left with the configuration of one consumer per queue, which can slow down performance and throughput.

Situations such as these present a unique opportunity for significantly increasing the overall performance and throughput of your messaging system. Rather than use the point-to-point model, simply switch to the pub/sub model. But wait you say—the pub/sub model is only used for broadcast-related messaging! Not true. As long as you limited yourself to a single subscriber and you have a good level of governance within your environment, pub/sub can be used in the same manner as point-to-point.

As you can see by the graph in Figure RIC-1, pub/sub messaging is significantly faster than point-to-point messaging, particularly for increased messaging loads. Part of the reason for this increase in performance is due to the routing nature of the pub/sub model (push vs. pull). With point-to-point messaging, the message broker has to store every message in a central queue (either in memory or on disk) and "assign" the message to a consumer by attaching a client ID to the message. The consumer then looks for the next message assigned to it and pulls the message from the queue, doing handshakes all along the way. All of that logic takes a fair amount of time. With the pub/sub model, messages are simply forwarded (pushed) to subscribers, therefore making it significantly faster than the point-to-point model.

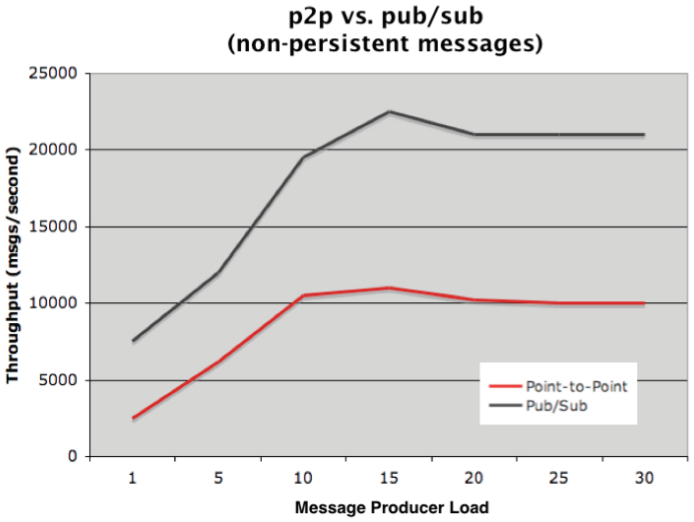

Figure RIC-1

Converting from the point-to-point model to the pub/sub model is fairly simple and straightforward. You will first need to change JMS API references from queue-based objects to topic-based objects as follows:

Queue → *Topic*

QueueConnectionFactory → *TopicConnectionFactory*

QueueConnection → *TopicConnection*

QueueSession → *TopicSession*

QueueSender → *TopicPublisher*

When sending a message, rather than invoking the *send* method on the *QueueSender*, you need to invoke the *publish* method on the *TopicPublisher*.

Probably the most intrusive change is converting your message consumer to a *subscriber*. When converting from the point-to-point messaging model to the pub/sub model, you will generally want to create a *durable subscriber* so that messages are preserved if the subscriber is not connected at the time the message is sent. To make this change, simply change your *QueueReceiver* object to a *TopicSubscriber* and create a durable subscriber by invoking the *createDurableSubscriber* method on the *TopicSession* object.

Request/reply messaging works the same way for pub/sub as it does for the point-to-point model. The *replyTo* message header property would contain a topic name rather than a queue name, and you would create a *TopicSubscriber* in your publisher code to receive the return message. The rest of the messaging aspects (message selectors, correlation IDs, and message IDs) remain the same.

Using the pub/sub model for standard business message processing may seem a little strange, but as long as you make sure no other subscribers are listening to the topic you essentially have the same type of configuration as the point-to-point model, only significantly faster.

Using Multiple Queues

As indicated in the first section, the typical approach for increasing the throughput of a messaging system using the point-to-point messaging model is to add multiple concurrent consumers to a queue, thereby making it possible to process multiple messages simultaneously. This approach is illustrated in Figure RIC-2.

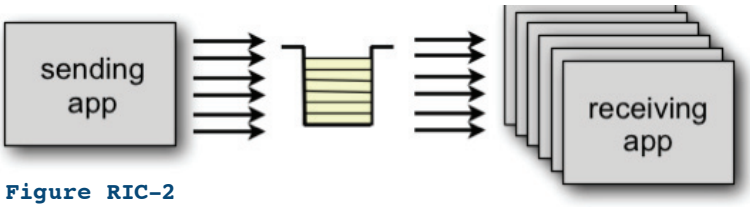

Figure RIC-2

Since message consumers take up memory, CPU, and other resources (e.g., database connections), there is a practical limit as to how many competing consumers you can have. Therefore, you will always arrive at a point where adding additional competing consumers actually *degrades* overall system performance and throughput.

Keeping the same number of competing consumers, you can significantly increase the throughput and performance of your messaging system by changing your configuration from one queue to multiple queues and distributing the competing consumers between the queues. Note that adding multiple queues does not resolve any resource issues you may encounter with respect to adding additional competing consumers, which is why the same number of competing consumers is used. This configuration is illustrated in Figure RIC-3.

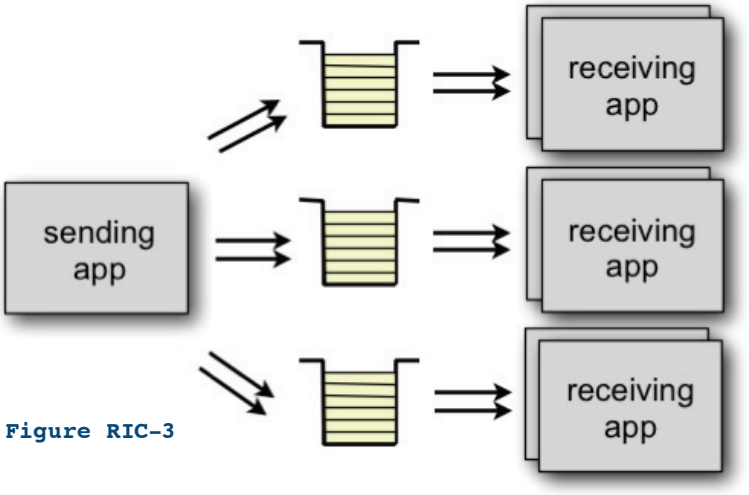

Figure RIC-3

With this configuration the performance difference becomes rather significant, particularly when processing a large number of messages. As illustrated in Figure RIC-4, the more messages you send through your messaging system, the greater the increase in throughput.

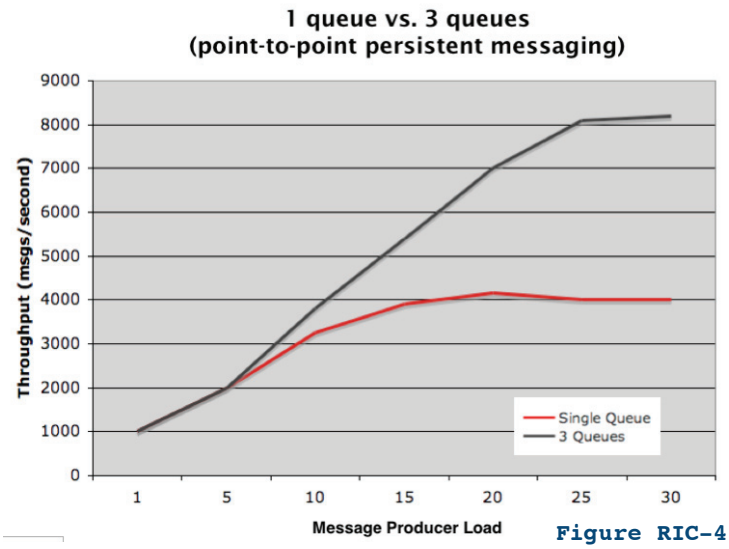

Figure RIC-4

To implement this technique you simply create a round-robin algorithm within your message producer that loops through an array of queues, selecting the next one whenever it sends a message. One way to do this would be to first create an array of queue names as in Listing RIC-3.

Then, when sending a message, you would calculate a queue index and use that index to get the next queue name in the array, as illustrated in the Spring JMS code in Listing RIC-4.

Of course, there are a variety of ways of doing this, but the main point here is that it is the responsibility of the message producer to maintain the list of queues and to select the next one to send a message to.

Paying Attention to Message Persistence

The default behavior in JMS when sending a message is to persist it by saving the messages to a file or a database prior to the consumer receiving it. One of the main reasons for doing this is to provide guaranteed delivery within your messaging system.

```
String[] queues = {"TRADE1.Q", "TRADE2.Q", "TRADE3.Q"};
```

Listing RIC-3

```
queueIndex = (queueIndex == queues.length-1) ? 0 : queueIndex+1;
jms.convertAndSend(queues[queueIndex], msg);
```

Listing RIC-4

Without message persistence you have the possibility of losing messages, particularly if you experience a system failure before messages are received and acknowledged by the consumer. However, message persistence (and hence guaranteed delivery) comes with a price.

As illustrated in Figure RIC-5, the performance difference between using persistent and non-persistent messages for point-to-point messaging is fairly significant, particularly as you increase the number of messages being processed by your system.

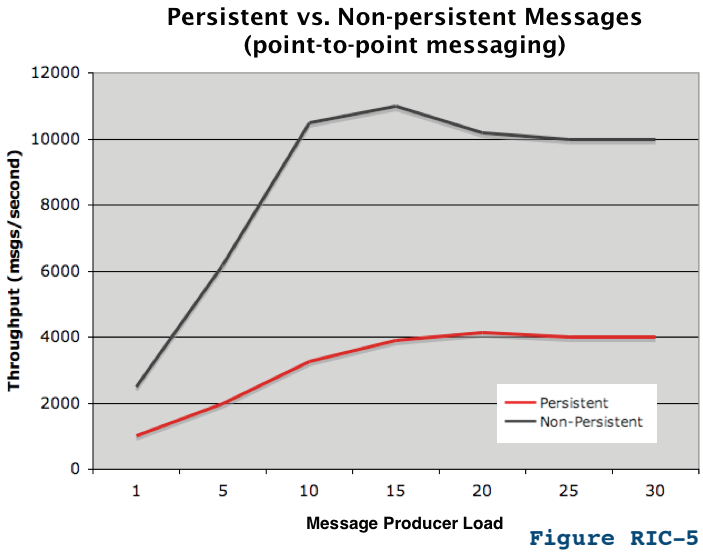

Figure RIC-5

When looking for performance improvements in your messaging system, you should always look at your message persistence strategy and question whether the messages you are sending really need to be persisted, keeping in mind the tradeoffs involved (guaranteed delivery vs. high performance).

For example, if you are persisting log or audit messages, consider whether the tradeoff of possibly losing messages on the (hopefully) rare chance of a system failure is worth the degradation in overall performance and throughput. The possibility of losing a few in-flight messages on a system outage might be worth the significant increase in performance.

Another good example is the case where you are processing a batch of messages from an incoming file. Assume that if the system goes down you need to back out any changes up to that point and resubmit the file for processing, or maybe even overwrite prior updates when the file is resubmitted for processing. In this case there is no need to persist messages because they are resubmitted and reapplied after a system failure.

You can control message persistence at the *QueueSender* level (which applies to all messages sent by that sender) or the individual message level as the message is sent. To specify that all messages sent by a sender should be non-persistent, you would set the delivery mode on the message header to *DeliveryMode.NON_PERSITENT* as illustrated in Listing RIC-5.

```
QueueSender sender = session.createSender(queue);
sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

Listing RIC-5

Alternatively, you can use the default delivery mode (*DeliveryMode.PERSITENT*) on the *QueueSender* and set the delivery mode for each message when it is sent as shown in Listing RIC-6.

```
QueueSender sender = session.createSender(queue);
sender.send(msg,DeliveryMode.NON_PERSISTENT,4,0);
```

Listing RIC-6

When it makes sense and is possible, this simple technique can significantly increase performance with very minor changes to your source code.

Conclusion

These four message performance techniques are relatively simple to implement and can significantly speed up the overall performance and throughput of your application. However, there are some tradeoffs to consider with respect to the increase in performance and throughput you will experience.

With the first technique (Spring min-max concurrent consumers) the tradeoff for better performance is higher resource utilization due to the increase in the number of consumers. The tradeoff for the second technique (leveraging pub/sub) is the risk of messages being processed multiple times due to multiple subscribers listening on the topic. The third technique (multiple queues) doesn't have any tradeoffs other than the need to maintain multiple queues in your messaging system (not highly significant). The tradeoff with the last technique (non-persistent messages) is the possibility of in-flight messages being lost in the event of a system failure.

You will need to analyze these tradeoffs to make sure that the increases in performance and throughput you will experience outweigh the negative aspects. If so, then get ready for lightning fast messaging performance with minimal effort.

About the Author

Mark Richards is a Director and Enterprise Architect at Collaborative Consulting, LLC (http://www.collaborative.com), a Boston-based Business and Architecture Consulting Firm, where he is involved in the architecture, design, and implementation of event-driven architecture, service-oriented architecture, messaging systems, and enterprise service bus technologies. Prior to joining Collaborative Mark was an Executive IT Architect with IBM, where he worked as an SOA and enterprise architect in the financial services area. Having served in the IT industry since 1984, he has significant experience in the architecture and design of small to large systems on a wide range of languages and platforms. He is the author of several technical books (including the recently published *Java Message Service, 2nd Edition* by O'Reilly), and has spoken at over 85 technical conferences worldwide. You can read more about Mark by visiting his website at http://www.wmrichards.com.