The Art of Messaging

by Mark Richards

Once you learn the Java Message Service (JMS) API and understand the basic configuration settings of your messaging provider, it's time to apply that knowledge towards designing efficient and robust messaging systems. This article will help steer you in the right direction and teach you how to avoid some of the more common pitfalls associated with messaging.

Messaging – a Science or an Art?

In a recent interview for a NFJS network webcast, Jay Zimmerman asked me whether I thought messaging was a science or an art. My answer was that I thought it was both. Messaging is a "science" in the sense of the JMS API syntax, basic messaging constructs and concepts, and learning how the providers work from an installation and configuration standpoint. However, messaging is also an "art" in the sense of how to design robust and effective messaging systems.

It's easy enough to learn the science part of messaging through books, Google searches, online examples, and even the user guides that come with most messaging-based products. However, learning the art of messaging, that is to say, the design of messaging systems, takes years of practice coupled with lots of trial-and-error. Knowing the implications of certain design decisions and knowing what works and doesn't work in various scenarios is what separates the expert from the journeyman.

Unfortunately, I can't teach you all there is to know about the art of messaging in this short article. However, what I can do is guide you in the right direction so that you can avoid some of the common messaging pitfalls and understand the implications of various messaging design patterns.

In this article I will discuss the advantages and disadvantages of using the all-too-common "Single Queue Design", how leveraging JMS message priority may not be the best solution for solving the priority and ordering of messages, and finally why it seems that some messages never expire, even though you have an expiration set on the message.

Single Queue Design vs. Multiple Queue Design

The "Single Queue Design" technique is a common design practice in which all messages are sent to a single queue, regardless of the content of each message. With this technique the message listener typically consists of a router component that routes the message payload to a non-messaging component for processing based on some sort of message type. The single queue design technique is illustrated in Figure RIC-1 "Single Queue Design".

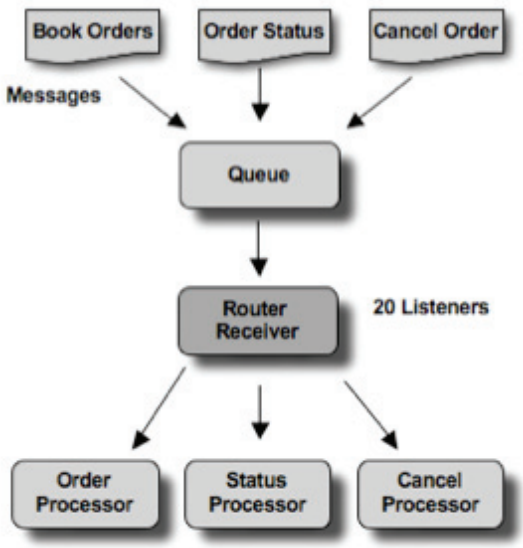

Figure RIC-1

The first thing you'll most likely notice from Figure RIC-1 "Single Queue Design" is the simplicity of the design. In this case book order messages, cancel order messages, and order status messages are all funneled through a single queue. The message router

listener receives the request and routes the XML-based payload to the *OrderProcessor* component, *StatusProcessor* component, or the *CancelProcessor* component (don't forget, with messaging these can be Java or non-Java components!).

The router message listener usually routes messages (or message payload in this case) based on a property in the header (for example, a custom message header property called *type*). Listing RIC-1 "Message Router" shows the typical structure for a common message router.

```
public void onMessage(Message msg) {
  //extract msg payload
  String xml =
    ((TextMessage)msg).getText();
  int type =
    msg.getIntProperty("type");

  //route msg payload
  if (type == BOOK_ORDER) {
    orderProcessor.placeOrder(xml);
  } else if (type == ORDER_STATUS) {
    statusProcessor.checkStatus(xml);
  } else if (type == CANCEL_ORDER) {
    cancelProcessor.cancelOrder(xml);
  } else {
    throw new RuntimeException
       ("Invalid Type");
  }
}
```

Listing RIC-1 – Message Router

The single queue design has a lot of (perceived) advantages. First, it provides for ease of maintenance. Adding another message type is simply a matter of defining a new XML schema, creating a new message processor, and adding an extra conditional statement in the message router. No work is needed on the messaging-side of things such as configuring additional queues. Second, it is a simple design and simple is usually good. There is only a single queue to administer and the system has plenty of capacity and throughput to handle all of the messages. Third, this design provides for a high degree of agility because there are fewer components (and hence fewer people) that need to be involved in changes (i.e., no additional queues need to be defined for additional message types). All of these advantages make the single queue design a fairly popular design choice for messaging systems.

There must be a catch, right? Indeed, there is. To begin with, the number of listener threads you can have listening on a single queue is not infinite. Message listener threads consume JVM and machine resources, as well as database connections and other potential external connections. In my experience it is usually the database rather than memory or CPU that restricts the number of concurrent message listeners. Therefore, assume for purposes of this discussion that 20 listeners (a typical number for a standard messaging application) is the upper limit for this particular system.

Although the single queue design approach looks like a good design technique, it can lead to poor request load balancing, poor performance, and poor user concurrency. In this design all messages are treated as equal. This means that book orders are treated with the same priority as order cancel requests or order status requests. It would hardly be in the bookseller's best interest to hold up book orders while other people are canceling their orders or checking their order status. As you will see in the next section, adjusting the message priority is not the right solution, and ends up making the situation even worse.

The other problem with this design approach is that there is no way to load balance resources between the various messaging types. In other words, there is no way to guarantee you can process book orders, order cancel requests, and order status requests concurrently. You may get a flood of order status requests using up the listener threads, while book order requests sit on the queue waiting to be processed.

A better design approach is to use a separate queue for each messaging request type, as is shown in Figure RIC-2 "Multiple Queue Design".

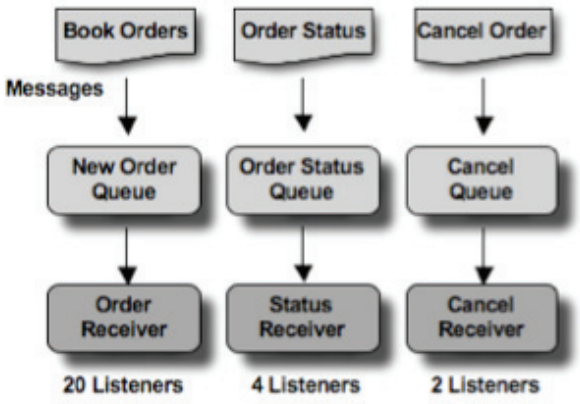

Figure RIC-2

With the multiple queue design illustrated in Figure RIC-2 "Multiple Queue Design", you can easily adjust the number of listener threads on each type of message listener to accommodate the anticipated load. In this example, there are 20 listener threads for book orders (those need to be processed as quick as possible), whereas, due to resource constraints and volumetric data, there are only 4 listener threads needed for the status requests and only 2 listener threads for order cancel requests (the reason being that if someone waits long enough maybe they won't cancel the order!). With this model, book orders, cancel requests, and status requests can all be processed concurrently and tuned independently from one another, making this a much more effective and efficient design approach. The multiple queue design approach also increases the overall throughput of your system because you have more fine-grained control over how the incoming messages are processed.

Using Message Priority

In the prior section I warned against using message priority as a way to manage the order and priority of messages in the queue. Wait a minute – isn't that what message priority is all about? It is, but when used for the wrong reasons it could lead to poor performance and throughput.

Consider a simple example where you have the same message type (e.g. order request) coming in from two different channels; one from a standard consumer-based website, and one from a commercial B2B (business-to-business) third-party batch feed. Because the web-based client channel request has already been authenticated and partially processed, they are relatively quick to process – 5 seconds to process the order. On the other hand, each of the orders in the commercial B2B channel require verification and preprocessing, resulting in 30 seconds of processing time for each order in the transmitted batch. This scenario is illustrated in Figure RIC-3 "Multiple Channels".

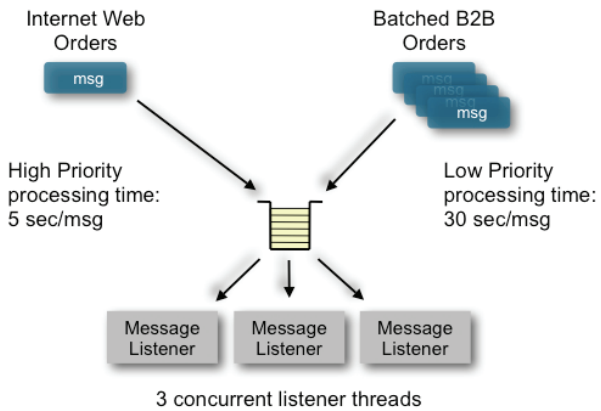

Figure RIC-3

Because you don't want to keep your web-based customers waiting, you set the message priority on web-based requests to high priority (9), indicating to the messaging provider that these messages should be processed first when in the queue. You also set the batched orders to low priority (1) so that they are processed last – after all, no one is waiting for those ones to be immediately processed.

While this technique seems to make perfect sense, all is not as it seems. You see, the problem is your customers are constantly complaining about 30+ second response times for web-based requests. Wait a minute – before, I said the web orders only take 5 seconds to process. What's going on? If you do the math, you will discover why this is a bad design practice. Here's the scenario:

1) At time t0, 20 batch orders arrive at low priority into the queue

2) At time t1, 3 batch orders are picked up by the listeners (don't forget - only 3 listener threads are available)

3) At time t2, 5 web orders arrive immediately after time `t1`

Because all of the available listeners are busy, they won't be available for another 30 seconds. Therefore:

Web Order 1 takes 35 seconds response time (30 seconds wait for a listener, 5 seconds to process)

Web Order 2 takes 35 seconds response time (30 seconds wait for a listener, 5 seconds to process)

Web Order 3 takes 35 seconds response time (30 seconds wait for a listener, 5 seconds to process)

Web Order 4 takes 40 seconds response time (35 seconds wait for a listener, 5 seconds to process)

Web Order 5 takes 40 seconds response time (35 seconds wait for a listener, 5 seconds to process)

Now, you might be tempted to increase the number of listener threads to address this problem, but by how many? Remember, there is a practical limit to how many listeners you can have on any given system. While increasing the listener threads to 25 might solve this particular scenario, what would happen if a batch of 100 orders came in during an idle state? How about 200 batch orders? How about a thousand batch orders? The point is, regardless of the number

of listener threads you have, at some point you will have to wait for the long-running batch orders to complete before you can process a fast running web-based request. While your particular scenario might be different, you can easily see the effects of relying on message priority as a means to control the flow of messages through the system.

A better design approach for this type of scenario is to use separate queues for each channel, even though you are processing the *exact same message*. This is similar to the discussion in the prior section – by having separate queues you can now tune each listener to handle a certain number of requests, and you can process web-based orders and B2B orders concurrently.

The next time you find yourself having to use message priority in your application, check to see if it would make more sense to have separate queues instead. Do the math, test different scenarios, and decide if using multiple queues is a better approach. In most cases, particularly under heavy load, using multiple queues is a much better approach over using message priority settings and a single queue.

Message Expiration

By default, messages in JMS are set to never expire. However, there may be times when you want to set a message expiry so that if the message cannot be delivered, it will be removed from the primary queue and sent to a special queue known as a Dead Letter Queue (DLQ).

Message expiration is controlled through the *JMSExpiration* message header property, with corresponding *setJMSExpiration()* and *getJMSExpiration()* methods for getting and setting this property. These methods are located in the *javax.jms.Message* object. A common pitfall is for application developers to set the message expiration by using the *setJMSExpiration()* method, as is shown in Listing RIC-2 "Incorrect Way for Setting Expiration".

Notice in Listing RIC-2 how the developer is trying to set the message expiration to 30 seconds by directly invoking the *setJMSExpiration()* method. In fact, based on the code above, this message will never expire! This is a very common pitfall that is easy to fall into.

```
//incorrect way to set
//the message expiration
...
TextMessage msg =
   Session.createTextMessage();
msg.setText(xml.toString());
msg.setJMSReplyTo(response);
msg.setJMSExpiration(
  new Date().getTime() + 30000);
sender.send(msg);
...
```

Listing RIC-2
Incorrect way for setting expiration

The reason this message will never expire is because the *message provider* invokes the *setJMSExpiration()* method when the message is being sent, overriding the method you invoked in the source code prior to sending the message (as in Listing RIC-2). The message provider will use the value passed in through the *timeToLive* property of the message object for the message expiration (assuming it is greater than zero). Since the default value of the *timeToLive* property is 0, the message will never expire.

There are two ways to correctly set the expiration for a message. The first technique is to use the *setTimeToLive()* method on the *QueueSender* or *TopicPublisher* object, which will then (by default) set the expiration for each message sent or published from that message producer. The second technique is to set the message expiration for each message individually when sending or publishing the message (via the *send()* or *publish()* method). These techniques

```
//Correct way to set
//the message expiration
...
TextMessage msg =
   Session.createTextMessage();
msg.setText(xml.toString());
msg.setJMSReplyTo(response);
sender.setTimeToLive(30000);
sender.send(msg);
...
```

Listing RIC-3
Correct way for setting expiration (1)

are illustrated in Listings RIC-3 "Correct Way for Setting Expiration (1)" and RIC-4 "Correct Way for Setting Expiration (2)" respectively.

```
//Correct way to set
//the message expiration
...
TextMessage msg =
   Session.createTextMessage();
msg.setText(xml.toString());
msg.setJMSReplyTo(response);
sender.send(
    msg,
    DeliveryMode.PERSISTENT,
    4
    30000);
...
```

Listing RIC-4
Correct way for setting expiration (2)

When setting the message priority for the individual message using the *send()* method (as illustrated in Listing RIC-4), you have to also specify the delivery mode (second parameter), the message priority (third parameter), and finally the message expiration.

Of course, you can certainly combine these techniques if you have a particular message out of the group that has a different expiration than what is specified at the *QueueSender* level. For example, assume that all messages by default have a message expiration of 1 hour. However, for some messages, you want to set the message expiration to 1 minute instead. You could accomplish this by first setting the *timeToLive* property on the *QueueSender* to 3600000 (1 hour), then for particular messages needing a shorter expiration time, override it in the *send()* method. This technique is illustrated in Listing RIC-5 "Overriding Default Expiration".

```
...
//all messages by this
//sender expire in 1 hour
sender.setTimeToLive(360000);
...
TextMessage msg =
   Session.createTextMessage();
msg.setText(xml.toString());
msg.setJMSReplyTo(response);
sender.send(msg);
...
//msg2 has a shorter expiration
TextMessage msg2 =
   Session.createTextMessage();
msg2.setText(xml2.toString());
msg2.setJMSReplyTo(response);
sender.send(
    msg2,
    DeliveryMode.PERSISTENT,
    4
    30000);
...
```

Listing RIC-5
Overriding default expiration

Conclusion

There are a lot of decisions that need to be made regarding the design of messaging systems. While some ideas like the "Single Queue Design" or using message priority seem like good ideas at the start, they can lead to poor performance and poor throughput down the road. Understanding these issues, as well as some of the pitfalls associated with messaging, will help you develop robust and efficient messaging systems.

References

Learn more about messaging by reading the book "Java Message Service, 2nd Edition" by Mark Richards (http://www.amazon.com/Java-Message-Service-Mark-Richards/dp/0596522045)

Read the JMS 1.1 Specification by going to http://java.sun.com/products/jms/

About the Author

Mark Richards is a Director and Senior Architect at Collaborative Consulting, LLC, a Boston-based Business and Architecture Consulting Firm, where he is involved in the architecture, design, and implementation of SOA, messaging, and other architectures, primarily in the Java platform. 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. He has been involved in the software industry since 1984 and has many battle scars to show for it. Mark served as the President of the Boston Java User Group in 1997 and 1998, and the President of the New England Java Users Group from 1999 through 2003. Mark is the author of the new book "Java Message Service (2nd edition)" from O'Reilly. He is also the author of "Java Transaction Design Strategies", contributing author of the new book "97 Things Every Software Architect Should Know" from O'Reilly, contributing author of "NFJS Anthology Volume 1", and contributing author of "NFJS Anthology Volume 2". Mark has many architect and developer certifications, including those from IBM, Sun, The Open Group, and Oracle. He is a regular conference speaker at the No Fluff Just Stuff Symposium Series and speaks at other conferences and user groups around the world. When he is not working Mark can usually be found hiking with his wife and two daughters in the White Mountains or along the Appalachian Trail.