Understanding the Differences between AMQP & JMS

by Mark Richards

As the Advanced Message Queuing Protocol (AMQP) specification matures and robust AMQP implementations such as RabbitMQ become more popular, you can't help but wonder whether you should jump on the bandwagon and use AMQP instead of Java Message Service (JMS) as your messaging standard. Understanding the differences between AMQP and JMS is a great way of understanding what AMQP is and whether you should use it. In this article I will describe AMQP through a comparison of the AMQP specification and the ever-popular JMS specification.

Introduction

For over 10 years the Java Message Service (JMS) has provided the Java community with messaging flexibility and standardization. In fact, many robust enterprise class open source and commercial message brokers now provide built-in message bridges that allow cross-platform messaging between the Java platform and other heterogeneous platforms and languages such as Ruby, C++, and C# (to name a few). Within the Java Platform you can also replace one JMS message broker with another with a few simple configuration changes to your Java applications. With all this flexibility and robustness, it certainly doesn't seem like the industry needs a new messaging standard–or does it?

The purpose of this article is to introduce you to some of the core differences between the Advanced Message Queuing Protocol (AMQP) and JMS so that you have a better understanding of what AMQP is and why it is such an important step in the evolution of enterprise messaging. You can then use this information to assist in deciding whether or not AMQP is the right choice for you.

A Brief History of AMQP

The original goal of AMQP was to solve the problem of messaging interoperability in the financial industry between heterogeneous platforms and message brokers (both commercial and custom). Developed by John O'Hara from JPMorgan and first used in production in 2006, AMQP set out to address the problem of interoperability by creating a standard for how messages should be structured and transmitted between platforms. This standard binary wire-level protocol for messaging would therefore allow heterogeneous disparate systems between companies (and also within companies) to easily exchange messages between each other, regardless of the message broker vendor or platform used by each company.

The AMQP specification is an open specification being developed by the AMQP working group, which currently consists of 20 companies in the financial, technical, and network marketing segments. You can get more information about the background of AMQP, as well as view or download the latest specification, by visiting the AMQP website at http://www.amqp.org.

Although AMQP may seem like one of those new, unproven technologies or the latest "flavor-of-the-month", it's not; there are currently hundreds of critical systems using AMQP as their messaging standard in a variety of market segments including finance, telecommunications, and defense. In addition, there are several open source implementations of AMQP currently available (with RabbitMQ, StormMQ, and Apache Qpid being the most popular), as well as several open source integration platforms that support AMQP messaging, including Mule, Apache Camel and Spring Integration.

The Big Picture

JMS is a very robust and mature specification that provides a standard messaging API for the Java Platform. With JMS you can replace any JMS-compliant message broker with another one with little or no changes to your source code (usually only configuration changes are needed). It also allows for interoperability between other Java Platform languages such as Scala and Groovy, and provides a level of abstraction that frees you from having to worry about specific wire protocols and JMS message broker vendors.

JMS defines a standard for interoperability within the Java Platform, but not outside of the Java Platform. Integration between the Java Platform and other platforms or languages (e.g., Java to Ruby), and even messaging interoperability between non-Java platforms (e.g., .NET to Ruby), is usually proprietary and can be tricky to design and implement.

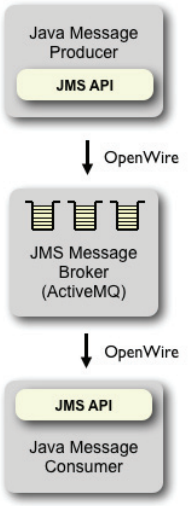

Figure RIC-1

For example, take the simplest case where you need to communicate between two Java messaging clients within the Java Platform. In this situation you would use the JMS API for both messaging clients and a JMS-compliant message broker (such as ActiveMQ) to store and forward messages. This scenario is illustrated in Figure RIC-1.

The example in Figure RIC-1 illustrates a typical JMS configuration. Notice that the protocol used to communicate between the messaging clients and the broker is OpenWire, the native wire protocol for ActiveMQ. In this scenario you can easily replace ActiveMQ with another JMS-compliant message broker (such as HornetQ from JBoss) with little or no coding changes. Instead of OpenWire, you would then be using the native protocol for HornetQ. The point here is that within the Java platform the protocol (or the message broker for that matter) doesn't really matter–its functionality is the same either way.

Now consider the case where you want to send a message from a Java message producer to a Ruby message consumer (or visa-versa). Since Ruby can't use JMS, you need a message broker that can bridge the two platforms and transform the protocol and message structure used by each platform. Since the most popular protocol choice for Ruby is the STOMP protocol (Streaming Text Oriented Messaging Protocol), you would need a message broker that can support both STOMP and JMS at the same time. This scenario is illustrated in Figure RIC-2.

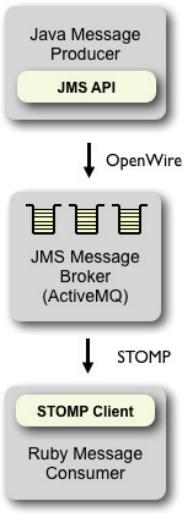

Figure RIC-2

In this case you can use an open source messaging provider such as ActiveMQ, which contains a built-in message bridge that can transform its native OpenWire protocol used by JMS into STOMP, and also convert the message structure from a JMS message to a STOMP message. Although this scenario seems to solve the cross-platform interoperability problem, it does have its disadvantages. First, both protocols may not support the same message body types (e.g., `ObjectMessage`, `StreamMessage`, etc.). Second, you would essentially be locked into one specific vendor solution (or in some cases only a few vendor choices) due to the built-in message bridge. Finally, both protocols may not support the same data types, custom properties, and header properties between the two messaging clients. Therefore, you can see that while cross-platform interoperability is certainly possible (as illustrated in Figure RIC-2), it is restrictive, limited, and forces vendor lock-in.

Now take this example one step further and consider the case where you need to provide message interoperability between a .NET C# messaging client and a Ruby messaging client. Assume the .NET C# messaging client is configured to use the NMS API (.NET Message Service API) with the MSMQ protocol, where as the Ruby messaging client is configured to use a STOMP client with the STOMP protocol. This scenario is illustrated in Figure RIC-3.

Figure RIC-3

Notice the big question mark in the middle of Figure RIC-3—that is where the message broker should be. In this case you would need to find a middleware vendor that provides a message bridge between MSMQ and STOMP and can transform both the protocol and message between the two disparate systems. Even if you did find a middleware vendor that supported both

protocols, chances are you would still need to change source code, client libraries, and configuration settings for each of the messaging clients.

I could provide dozens of other examples using combinations of other platforms, protocols, and message providers, but it isn't necessary; these scenarios clearly illustrate that JMS isn't the right solution for cross-platform interoperability. While it is true vendors such as SonicMQ, ActiveMQ and HornetQ provide some level of cross platform interoperability, they do so through proprietary protocols, APIs, and client libraries. Without some sort of messaging standard, it is difficult to know what messaging system another company is using and how to interface with it.

The need for a messaging-based cross-platform interoperability standard is exactly why AMQP was created. Whereas JMS provides a standard *messaging API* for the Java Platform, AMQP provides a standard *messaging protocol* across all platforms. AMQP does not provide a specification for an industry standard API. Rather, it provides a specification for an industry standard wire-level binary protocol to describe how the message should be structured and sent across the network.

So what API and corresponding client library should you use for AMQP? Which AMQP message broker should you use? Guess what—*It doesn't matter*. With AMQP, you can use whatever AMQP-compliant client library you want, and any AMQP-compliant broker you want (they do not have to be the same). This means that message clients using AMQP are completely agnostic as to which AMQP client API or AMQP message broker you are using.

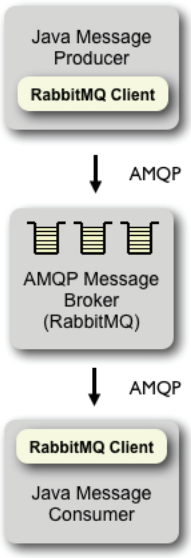

Figure RIC-4

To illustrate this important point, consider the previous three interoperability scenarios from before. In the Java-to-Java scenario, you could use an AMQP message broker such as RabbitMQ in place of ActiveMQ and use the RabbitMQ API as illustrated in Figure RIC-4.

Notice in Figure RIC-4 you are no longer using the JMS API[1]. Because these Java messaging clients are using AMQP, they are now able to seamlessly send and receive messages to and from other platforms and languages that also use AMQP.

To further illustrate this point, consider the prior Java-to-Ruby scenario using ActiveMQ. Remember how you had to be concerned about protocols and what message broker vendors supported both protocols? With AMQP you would simply use any AMQP-complaint client library (e.g., Apache Qpid) to connect to the AMQP message broker (e.g., RabbitMQ). In this scenario the Java Message Producer would use the RabbitMQ client API whereas the Ruby Message Consumer would use the Apache Qpid client API. Both would then be able to connect to the same AMQP message broker (in this case, RabbitMQ). This scenario is illustrated in Figure RIC-5.

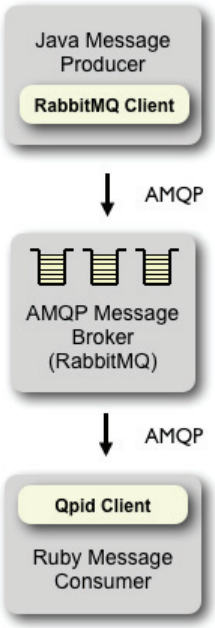

Figure RIC-5

The advantages of having an industry-wide messaging standard become very clear when considering these examples. A messaging system using AMQP as its messaging standard (regardless of the client API or message broker) can send messages back and forth to other messaging systems that use AMQP. In the example illustrated in Figure RIC-5 you could easily replace RabbitMQ with StormMQ, or the Java message producer with a C# message producer. It doesn't matter, as long as all the systems are using AMQP.

As you can see, AMQP has taken the broker-agnostic benefits of JMS within the Java Platform and escalated that concept to all platforms. Vendor messaging client libraries and vendor message brokers that support the AMQP protocol now become a commodity, allowing heterogeneous systems to easily communicate with one another through messaging, regardless of the platform or middleware vendor. This has profound implications for industries such as finance, telecommunications, defense, retail B2B, and other industries that need to communicate via messaging between systems in other companies.

1 As a side note, there is an effort currently underway by the AMQP Working Group to define a standard specification for JMS to AMQP mapping

Message Routing

Message routing describes how messages sent from a message producer (sender) get to the intended message consumer (receiver). There are some significant differences between AMQP and JMS in terms of how each handle message routing and the delivery of a message from a producer to a consumer.

JMS uses a simple message routing scheme where both the message producer and message consumer exchange messages by connecting to the same named queue or topic. For example, a message producer might send a message to a queue named "foo_queue" as follows:

```
queue = session.createQueue("foo_queue");
sender = session.createSender(queue);
msg = session.createTextMessage("test");
sender.send(msg);
```

A message consumer would then connect to the same named queue ("foo_queue") as the message producer and therefore receive the next message in that queue:

```
queue = session.createQueue("foo_queue");
receiver = session.createReceiver(queue);
msg = receiver.receive(10000);
```

Adding a message selector to the message consumer can further control routing in JMS. When using message selectors, even though both the message producer and the message consumer are connected to the same named queue, the message consumer would only receive messages if the message selector (in this case state = 'MA') evaluates to true:

```
queue = session.createQueue("foo_queue");
receiver = session.createReceiver(queue);
msg = receiver.receive(10000, "state = 'MA'");
```

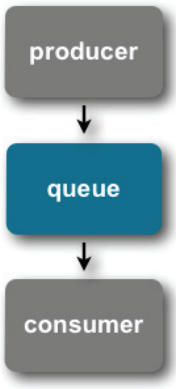

Figure RIC-6

The message routing scheme for JMS is illustrated in Figure RIC-6.

AMQP handles message routing a bit differently than JMS[2]. Rather than the message producer sending a message to a queue, it sends a message to an *exchange* along with a *routing key*. Exchanges are bound to queues in AMQP through what are called *bindings*. A binding is a directive indicating what messages should be routed from an exchange to a queue. Message consumers attach to a queue and receive messages from the queue that is bound to an exchange. If the routing key sent with the message matches the binding specified between the exchange and the queue, then the message is routed to the queue and consumed by the message consumer. This interaction is illustrated in Figure RIC-7.

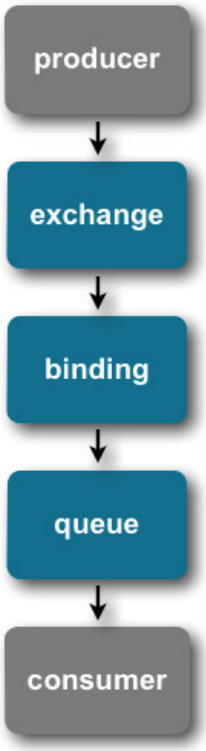

Figure RIC-7

To illustrate the interaction between the exchange, binding, and queue, suppose the diagram in Figure RIC-7 were setup in such a way that the queue shown in the diagram was bound to the exchange with a binding value "orders.us.ma". This means all orders in the U.S. for Massachusetts would be routed to the queue from the exchange. If a message is sent to the exchange from a message producer with a routing key value of "orders.us.ma", then it would be routed to the queue and the message consumer would receive the message. However, if a message were sent to the exchange with a routing key value of "orders.us.ca", it would not be delivered to the queue since the routing key and the binding do not match.

For example, to send a message to an exchange using RabbitMQ, a Java-based message producer would send a message to a named exchange called "orders" with a specific routing key value of "orders.us.ma" for that message as follows:

```
channel.exchangeDeclare("orders", "direct", true);
String routingKey = "orders.us.ma";
byte[] msg = "test".getBytes();
channel.basicPublish("orders", routingKey, null, msg);
```

To receive the message, the message consumer would first declare a queue ("ma_orders") and then bind that queue to the exchange named "orders" with a binding

2 The following description focuses on version 0-10, the latest specification as of the writing of this article - version 1-0 of the AMQP specification, which is currently in recommended status, handles message routing a little bit differently than is described here.

value matching the routing key ("orders.us.ma"). Since the binding specified in the message consumer matches the routing key specified for the message in the message producer, it would consume the next message on the queue:

```
String queue = "ma_orders";
channel.queueDeclare("ma_orders",…);
channel.queueBind(queue, "orders", "orders.us.ma");
channel.basicConsume(queue, …);
delivery = consumer.nextDelivery();
```

Notice that the message consumer has the responsibility of binding the queue to the exchange, as is shown in the preceding code with the `queueBind()` method. Interestingly, in AMQP the message producer has no knowledge of the queue that the message is being routed to nor the specific binding used. It is in the message consumer that you would specify the binding of the exchange to the queue. This is commonly referred to as *consumer-driven messaging*. If the binding in the preceding code were changed to "orders.us.ca", then the message consumer would not receive the message, even though the queue is bound to the exchange.

The routing model in AMQP essentially separates the transport model from the queuing model and therefore allows for complex and sophisticated routing capabilities as part of the specification. Multiple queues can be bound to an exchange with different bindings, thereby creating a context-based routing capability within the message broker.

Message Model Differences

There are two messaging models supported by JMS: Point-to-Point (queues) and Publish-and-Subscribe (topics). With Point-to-Point messaging, when you send a message to a queue it will be received by only one message consumer. With the Publish-and-Subscribe messaging, when you publish a message to a topic it will be received by all message consumers that subscribe to that topic.

AMQP offers five different messaging models called *exchange types*. The two mandatory exchange types that must be implemented by AMQP middleware vendors are the Direct Exchange and the Fanout Exchange. Optionally, a middleware vendor may choose to implement the Topic Exchange, Headers Exchange, and System Exchange.

The AMQP Direct Exchange closely matches the Point-to-Point messaging model in JMS. With the Direct Exchange, an exchange that is bound to a queue requires a direct match between the routing key and the binding in order for the message to be delivered to the message consumer. An important difference between these two models, however, is that with the Direct Exchange you can bind multiple direct queues to the exchange (meaning it is possible for more than one consumer to receive the message), whereas with the JMS Point-to-Point messaging model you are guaranteed that one and only one message consumer will receive the message.

The Fanout Exchange, Topic Exchange and Headers Exchange in AMQP are all essentially the same as the JMS Publish-and-Subscribe model. With the Fanout Exchange, queues bind to the exchange without a binding argument. When messages are sent to the exchange, they are unconditionally routed to any queue bound to that exchange. This is equivalent to the JMS Publish-and-Subscribe model without the use of message selectors.

With the Topic Exchange, queues bind to the exchange with a binding argument in the same manner as the Direct Exchange, but with the use of a wildcard (* or #). For example, a binding of "orders.us.ma.*" indicates that messages containing orders in the U.S. for any city in Massachusetts should be routed to the queue. Thus, if a message contains a routing key of "orders.us.ma.boston" it would be routed to the queue, whereas a message containing a routing key of "orders.us.ca.hollywood" would not be routed to the queue. This is equivalent to the JMS Publish-and-Subscribe model with the use of message selectors on the message consumer (subscriber).

The Headers Exchange is another close match to the JMS Publish-and-Subscribe model. Instead of using a routing key, the Headers Exchange uses message header values to determine a binding match for routing eligibility. For example, the binding for the Headers Exchange might be something like "state=MA, city=Boston, match=all". If the message header contains all of these properties and the values of each of those properties match the binding, then the message is routed to the queue. Like the Topic Exchange, this is equivalent to the JMS Publish-and-Subscribe model with the use of message selectors on the message consumer (subscriber).

The optional System Exchange in AMQP is a special kind of exchange that routes messages from an

exchange to an application service or system service rather than a queue. There is no real equivalent in JMS for this exchange type.

Message Structure Differences

The structure of a message in AMQP is very similar to that of JMS. In JMS a message is divided into three main sections – a *header section* that contains mostly immutable JMS header properties (e.g., JMSMessageID, JMSTimestamp, etc.), a *properties section* that contains mutable application-defined name-value-pair properties (e.g., state='MA'), and finally a *message body section* that can consist of one of five defined message types (Object, Map, Text, Bytes, and Stream).

AMQP also divides a message into similar sections (header, properties, body, and optional footer), but there are some differences from JMS. In AMQP, the header section of the message contains immutable application-defined properties, whereas the properties section of the message contains immutable routing and metadata properties (backwards from how JMS stores this information). The point here is that although there are some differences in the message structure between AMQP and JMS, they are similar enough not to have to worry about it much.

The only other message structure difference worth noting is that with AMQP there is only one message body type—a binary (bytes) message. With JMS there are five message body types. There is an effort underway within the AMQP working group to define a specification for JMS-to-AMQP mapping that vendors can use so that the message broker can transform the message from one of the five supported JMS message body types to an AMQP binary message and vice-versa. This specification can be found at https://amqp.org/svn/amqp/branches/amqp-1-0-jms-mapping. Regardless, it is always a good idea when using JMS to use either the `TextMessage` or more preferably `BytesMessage` to maintain portability.

Conclusion

By defining a wire-level protocol standard for messaging, the AMQP specification essentially creates a message-based interoperability model that is completely client (API) or server (message broker) agnostic. This means that as long as you are using AMQP you can connect and send messages to any AMQP message broker using any AMQP client.

My goal in writing this article was not to convince you to immediately jump onto the AMQP bandwagon, but rather to provide you with more information in order for you to build proper selection criteria to help you decide what you should use for your messaging standard. For example, if your message-based interoperability requirements are completely self-contained within the Java Platform and you have no foreseeable need to send or receive messages to and from other platforms, then it might be wise to stick with JMS for the time being until you have a need for cross-platform interoperability. If, however, you foresee (or have) the need to exchange messages between heterogeneous platforms, or if you are frequently struggling with messaging interoperability with other systems or other outside companies and want to remain completely vendor-agnostic with respect to your messaging systems, then AMQP is probably the right solution for you.

AMQP is an exciting next step in the evolution of messaging interoperability, and as such it significantly raises the bar in terms of using messaging for cross-platform interoperability. I am confident, as are most people in the messaging world, that AMQP will soon become the new messaging standard across all platforms.

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.