

Advanced Message Queuing Protocol

Steve Vinoski • IONA Technologies



I've often wondered why customers and vendors of certain types of middleware demand open standards, whereas no similar demand is evident for standards in other middleware areas. For example, much of the work I've done in my career has been in the distributed objects space, where synchronous messaging is the norm. Various middleware standards exist for synchronous messaging, including Corba's Internet Inter-ORB Protocol (IIOP), Java Remote Method Invocation (RMI), and SOAP. These protocols' purveyors and providers typically go the extra mile to ensure that their implementations interoperate cleanly with other implementations. They do this by exchanging implementations and performing their own in-house testing, or by attending special interoperability workshops at which everyone brings their code and tests interoperability with all other attendees' implementations. Such workshops are conducive to quickly achieving widespread interoperability between multiple implementations because, with code in hand, developers can often fix on the spot any interoperability issues that arise. Because customers often integrate multiple products that implement these protocols, they demand interoperability from the vendors who supply them.

The same doesn't seem to hold true in the world of asynchronous messaging, however, in which several proprietary products exist and use their own closed protocols. Examples of such systems include IBM Websphere MQ (formerly known as MQ Series) and Microsoft Message Queuing (MSMQ). One standard that these and others could follow is the Java Message Service (JMS) specification, which is arguably the best-known standard in the asynchronous messaging world. However, it's merely an interface, or API, standard: because JMS doesn't specify a standard protocol, JMS implementations provide their own, which are also effectively proprietary. Furthermore, JMS is limit-

ed to Java, which is only one viable implementation technology within the messaging middleware domain. The general lack of asynchronous messaging protocol standards means messaging implementations don't interoperate.

I'm unaware of any technical reasons for the asynchronous messaging world to be devoid of protocol standards, although I can think of plenty of nontechnical reasons. For example, an asynchronous messaging vendor might use the proprietary nature of its protocol as a way to achieve "lock-in" with its customer base. Alternatively, vendors selling both hardware and software might view proprietary messaging products as a means of selling more boxes. Or perhaps no customer has ever requested or stepped up to help create such standards, although that seems unlikely, given the overall affinity for standards that middleware customers and users generally display.

Fortunately, this is now changing. In June 2006, JPMorgan Chase (JPMC), Cisco Systems, Envoy Technologies, iMatix Corporation, IONA Technologies, Red Hat, TWIST Process Innovations, and 29West together announced the formation of the Advanced Message Queuing Protocol (AMQP) working group. The group's goal is to create an open standard for an interoperable enterprise-scale asynchronous messaging protocol.

What is AMQP?

Quite often, such announcements are just marketing fluff that generates heat with little light, quickly dying down without creating anything tangible. In this case, however, AMQP has several advantages right out of the gate:

- Rather than following the typical approach of starting from scratch and developing a specification based on "design by committee," the AMQP working group is beginning with a fair-

- ly complete protocol specification.¹
- In contrast to the typical waterfall-oriented standards approach of waiting until the specification is finished before anyone even attempts to implement it, multiple implementations of the current AMQP specification already exist – some are already in production. With the waterfall approach, developers usually find numerous inconsistencies and implementation warts in the “finished” specification once they start implementing it, thus inspiring a flurry of follow-on standards activities and specification revisions, or even simply ignoring the new standard such that it withers away.
 - One of the AMQP implementations is the open-source Qpid project, which is currently a “podling” project under the Apache Software Foundation Incubator (<http://incubator.apache.org/projects/qpid.html>). In terms of organizational boundaries, Qpid is wholly separate from the AMQP working group, although some individuals (including me) are involved in both. Qpid implements only publicly available specifications from the AMQP working group. It began life with code donated by JPMC and Red Hat, and by the time it entered the incubator, the code already provided operational message brokers in Java and C++, as well as supported messaging applications written in Java, C++, Python, and Ruby.

To date, most of the AMQP architecture has been driven by the finance community’s technical needs. This community uses services such as trading systems and banking systems that are among the most technically challenging and stringent distributed computing systems in the world, often requiring extremely high levels of performance, throughput, scalability, reliability, and manageability. In such systems, mere tenths of a microsecond can mean the difference between mak-

ing a successful trade ahead of the competition or missing it altogether. Reliability is also crucial because losing messages is simply not an option when a single message can represent a multimillion- or billion-dollar transaction. The AMQP contributors’ significant real-world experience in successfully building and deploying such systems, particularly those contributors from JPMC and iMatix, is another factor that helps set AMQP apart from the average run-of-the-mill messaging protocol.

Avoiding the Monolith

AMQP comprises both a network protocol, which specifies what client applications and message servers must send over the wire to interoperate with each other, and a protocol model, which specifies the semantics an AMQP implementation must obey to be interoperable with other implementations. Asynchronous messaging systems typically conjure up images of centralized, monolithic first-in, first-out (FIFO) queuing systems that accept messages at one end and dispense them from the other. AMQP takes a more modular approach, dividing the message brokering task between *exchanges* and *message queues*:

- An exchange is essentially a router that accepts incoming messages from applications and, based on a set of rules or criteria, decides which queues to route the messages to; exchanges don’t store messages.
- A message queue stores messages and sends them to message consumers. The storage medium’s durability is entirely up to the message queue implementation – message queues typically store messages on disk until they can be delivered, but queues that store messages purely in memory are also possible.

Joining together exchanges and message queues are bindings, which specify the rules and criteria by which

exchanges route messages. Specifically, applications create bindings and associate them with message queues, thereby determining the messages that exchanges deliver to each queue.

There are a couple of important points to note about the AMQ protocol model. First, the *chain of responsibility* pattern² is clearly in use. In this pattern, messages that appear to flow directly from sender to receiver actually flow through a set of message processors residing between the two. Each processor acts on the message along the way, perhaps adding to it, modifying its form, rejecting it, or simply passing it through to the next processor. The chain of responsibility pattern – quite common in modern middleware and distributed systems – enhances system flexibility by letting developers separate and combine orthogonal functionality as needed, thus avoiding static monolithic implementations. As of this writing, the AMQP working group is discussing using the chain of responsibility pattern to further augment the flexibility of bindings – allowing application-specific code to be invoked as part of a binding chain, for example.

The second important point to note about the protocol model is that it enables the broker to effectively make routing decisions. This contrasts with typical messaging systems, in which the logic for deciding which queue to deliver to or retrieve from is embedded within the applications that use the queues. Changing the message-routing and delivery logic for such systems therefore requires developers to touch all the affected applications; with AMQP, on the other hand, they can make such modifications by simply changing bindings on the queues themselves.

On the Wire

The other half of AMQP, the network protocol itself, ensures that implementations can successfully communicate and interoperate by speaking

the same “language” on the wire. By and large, AMQP is a straightforward protocol. It follows well-understood practices for data framing, option negotiation between client and server, and connection handling. AMQP currently assumes a stream-based transport (normally TCP) underneath it. It transmits sequential frames over channels, such that multiple channels can share a single TCP connection. Each individual frame contains its channel number, and frames are preceded by their sizes to allow the receiver to efficiently read them.

AMQP is a binary protocol. Debate about text-based versus binary protocols often rages in the general middleware and distributed systems communities. Binary protocols can pack much more data into a packet or frame, thus making them more efficient for applications like messaging for which data throughput is important. The typical argument for text-based protocols, on the other hand, is that they’re easier to code and debug. Personally, I find such arguments feeble because any programmer worth his or her salt should be able to easily encode and decode binary data for debugging purposes. Moreover, forcing all applications that use a given protocol to forever be less efficient just to make the protocol programmer’s life easier seems like the wrong trade-off.

AMQP’s application-level aspect specifies several protocol commands. To associate these commands with the entities they act on, the specification calls the commands *methods* and groups them into *classes*. With the `Exchange` class, for example, `Exchange.Declare` and `Exchange.Delete` are methods for declaring and deleting an exchange, respectively. Other classes include `Connection`, `Channel`, and `Queue`, as well as several classes for message content. As you’d expect, `Channel` supports `Channel.Open` and `Channel.Close` methods, which do as their names suggest. The `Queue` class supports `Queue.Declare`, `Queue.`

`Bind`, and `Queue.Delete` methods. The `Basic` class provides methods for normal message processing, such as `Basic.Publish` for sending messages, `Basic.Get` for synchronously taking messages off a queue, and `Basic.Ack` for acknowledging messages. Note that all the classes described here support additional methods, and various other classes and methods exist as well. However, those I’ve described here should be enough to show that the AMQP protocol command design is pretty straightforward.

AMQP Applications

The actual API through which applications interact with AMQP implementations depends on the programming language the developer uses. For example, you can map AMQP’s capabilities such that Java applications can use them through JMS APIs. For C++, Python, and Ruby applications, however, there are no popular open messaging API standards like JMS for Java, so those languages support their own AMQP APIs, which typically reflect the AMQP application-level protocol classes and methods.

Given that several of my previous columns have discussed the role of dynamic languages in middleware, it’s worth noting that the Python and Ruby AMQP bindings allow for significant source code brevity. By their very nature, these languages support rapid prototyping, allowing developers to quickly create small applications that let disparate applications communicate with each other. They’re also useful to those of us developing for the Qpid project, as they let us quickly write system tests that are sharable across all languages.

Shortly after the June AMQP announcement, I noted several technical discussion sites and blogs that questioned why anyone would need AMQP when the popular Extensible Messaging and Presence Protocol

(XMPP; www.xmpp.org) – the IETF formalization of the open-source Jabber protocol (www.jabber.org) – already exists. In reality, AMQP and XMPP have little in common in terms of the applications they serve. I believe the confusion around these protocols stems simply from the fact that they’re both forms of messaging protocols. However, the two differ vastly in the type of messaging each performs.

XMPP, as its name implies, is about presence. People use it primarily for instant messaging. AMQP, on the other hand, is about enterprise messaging. As explained earlier, enterprise messaging applications often require high levels of performance, throughput, scalability, and reliability. XMPP/Jabber is simply not intended for use under the extreme operating conditions that AMQP is designed to handle.

If you’d like to learn more about AMQP or even contribute to developing an implementation of it, please consider joining the Qpid developer community. AMQP is finally addressing the lack of enterprise messaging interoperability standards. This relatively simple yet compellingly powerful enterprise messaging protocol is thus poised to open up a bright new era for enterprise messaging. □

Acknowledgments

Thanks to Carl Trieloff of Red Hat for reviewing a draft of this column.

References

1. *AMQP: Advanced Message Queuing*, version 0.8, AMQP working group protocol specification, June 2006; available from www.ionac.com/opensource/amqp/amqp0-8-june19.pdf.
2. S. Vinoski, “Chain of Responsibility,” *IEEE Internet Computing*, vol. 6, no. 6, 2002, pp. 80–83.

Steve Vinoski is chief engineer for IONA Technologies. He’s been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the W3C. Contact him at vinoski@ieee.org.