Toward a Commodity Enterprise Middleware

JOHN O'HARA, JPMORGAN

AMQP (Advanced Message Queuing Protocol) was born out of my own experience and frustrations in developing front- and back-office processing systems at investment banks. It seemed to me that we were living in integration Groundhog Day—the same problems of connecting systems together would crop up with depressing regularity. Each time the same discussions about which products to use would happen, and each time the architecture of some system would be curtailed to allow for the fact that the chosen middleware was reassuringly expensive.

From 1996 through to 2003 I was waiting for the solution to this obvious requirement to materialize as a standard, and thereby become a commodity. But that failed to happen, and I grew tired of waiting.

Consequently, AMQP was created and had its first mission-critical deployment to production in mid-2006. That project paid for itself with its first deployment, serves 2,000 users, and processes 300 million messages per day.

This article sets out the motivations, capabilities, and credentials of AMQP and offers it as a practical solution for a standards-based messaging infrastructure.

AMQP is a binary wire protocol and well-defined set of behaviors for transmitting application messages between systems using a combination of store-and-forward, publish-and-subscribe, and other techniques.¹ I use the term *application messages* to distinguish AMQP from instant messaging or other forms of end-user messaging. AMQP

addresses the scenario where there is likely to be some economic impact if a message is lost, does not arrive in a timely manner, or is improperly processed.

The protocol is designed to be usable from different programming environments, operating systems, and hardware devices, as well as making high-performance implementations possible on various network transports including TCP, SCTP (Stream Control Transmission Protocol), and InfiniBand.

THE NEED FOR A STANDARD

Every major investment bank on Wall Street has at some point built its own messaging middleware. Many have either faded away or spun off to become commercial proprietary solutions.

Why do they build their own middleware? The financial services industry has some of the most demanding needs for messaging both in guaranteed delivery and in publish-subscribe. Demands often exceed the capabilities of currently available software, and there is no shortage of technology expertise in banks. Building one's own middleware is therefore a credible approach.

Banks are looking for high-performance service buses from which to hang their system architectures. Web services are not fitting the bill because they are too compute- and bandwidth-intensive per unit of work.

The growth of automated trading is also igniting inter-

The background is a vibrant red-to-orange gradient. It features a complex network of white and light orange geometric shapes. Large, semi-transparent circles of various sizes are scattered across the frame. A prominent, thick white arc curves from the bottom left towards the center. A series of small white circles are connected by thin white lines, forming a path that moves from the top right towards the center. On the right side, there are several vertical columns of small white dots. Additionally, there are three upward-pointing arrows: one large orange arrow in the upper right, and two smaller white arrows below it, one of which is partially obscured by a circle. Faint, illegible text is visible in the background, appearing as a watermark or bleed-through from another page.

Can AMQP enable a new era in messaging middleware?

Toward a Commodity Enterprise Middleware

est in improving middleware. Banks are still pushing the envelope with market data events exceeding 500,000 per second at source, according to the Options Price Reporting Authority. Processing that flood of information and executing timely business transactions off the back of it is challenging. Market data volumes exacerbate transaction-processing volumes.

Given the clear need, why do many internal efforts not endure? Despite their technical abilities, banks are not software houses; messaging middleware is complex software, and it is difficult for a bank to focus the level of intellect and talent on the problem for a long time.

Banks have managed to work together in creating open technical standards where such standards are absolutely necessary to doing business; the FIX (Financial Information Exchange) protocol, FAST (FIX Adapted for Streaming), FpML (Financial products Markup Language), and SWIFT (Society for Worldwide Interbank Financial Telecommunication) are all good examples.

In 2003 I embarked on a quest to standardize MOM (message-oriented middleware) technology *outside* my firm, so we could pick it up inside the firm and use it between firms.

MAKING IT HAPPEN

This had to be an industry initiative. Home-grown middleware could not thrive in the small market available within a host organization, even the largest host.

It is also notable that pervasive networking standards such as Ethernet, the Internet Protocol, e-mail, and the Web share some traits. They are all royalty-free and unencumbered by patents, they are all publicly specified, and they all shipped with a useful early implementation for free. The combination of freedom and usefulness drives their adoption when predicated on fitness for purpose.

To succeed, AMQP needed to adopt these same characteristics:

- It needed to be a fully defined, open, royalty-free, unpatented specification to enable anyone to implement a compatible service. Furthermore, the standard specification had to be clearly separate from the implementations;

otherwise, it would not be a fair market for commercial entities to enter. AMQP had to be appealing for commercial implementation and exploitation or it would not succeed.

- AMQP needed to have real implementations of the specification; otherwise, the specification would not be immediately useful or interesting to front-line developers with pressing needs. Ideally, it should have more than one implementation to qualify as a potential IETF draft standard.² So, there are real implementations you can run today (as detailed later in the article).

- AMQP software had to be proven in live systems. Middleware is a critical piece of any system and must be trusted. That trust has to be earned. To this extent, it was clear we would have to deploy an implementation in a high-profile, mission-critical application to assuage the fears of other early adopters. So, a combination of OpenAMQ and Qpid are live at JPMorgan, supporting 2,000 users on five continents and processing 300 million messages per day.

- Finally, and most importantly, AMQP needed to be a collective effort. Openness to partnership and the ideas of others had to be there from the beginning. To this end, we carefully selected a partner to co-develop the specification and implement the software. We chose iMatix, a boutique European development house that had clearly demonstrated a commitment to open source and sound ethics, and had a strong engineering background and excellent writing abilities.

Because the project was sponsored by a bank, it also had to “wash its own face,” as they say. This was *not* a research project. Through sheer good luck, there was a need to refresh some large systems with very specific requirements. This provided a tangible return for AMQP investment, so I was able to convince a forward-looking CIO that AMQP was the way to go.

THE AMQP WORKING GROUP

When the shape of AMQP had been worked out between JPMorgan and iMatix, and the basics of the specification forged in the heat of an initial implementation, the time was right to extend the partnership and encourage others to bring their talents to the specification and share ownership of AMQP's future.

STRONG GOVERNANCE

The heart of openness and trust in any group effort is effective governance. Expanding the group required a new contractual framework and a plan for the end game where AMQP could become a standard. Red Hat took the

lead in establishing the legal framework for the standard; it, too, understood the issues in managing open intellectual property. The key part of doing this is to ensure that everyone contributing has the authority to do so and that there is a paper trail from every potential owner of IP through to the group effort, and that the intent to share is clear even in draft revisions of specifications. The result was a contract that clearly committed the members of the working group to promote unrestricted open middleware through AMQP.

The members of the working group have granted licenses to the necessary parts of their patent portfolios to anyone who wants to implement AMQP. You can see the license grant in the specification itself.

This level of contribution shows the commitment of the group to open middleware. The AMQP Working Group's Web site is <http://www.amqp.org>.

USER DRIVEN

The AMQP Working Group is quite unique in technology standards work because of the heavy involvement of users. JPMorgan, Credit Suisse, TWIST, and to some degree Cisco are more end users than developers. This balance leads to a group of people interested in solving the problem, not pandering to technology agendas or product agendas.

ARCHITECTURE

From the beginning, AMQP's design objective was to define enough MOM semantics (see figure 1) to meet the needs of most commercial computing systems and to do so in an efficient manner that could ultimately be embedded into the network infrastructure. It's not just for banks.

AMQP encompasses the domains of store-and-forward messaging, publish-and-subscribe messaging, and file transfer. It incorporates common patterns to ease the traversal of firewalls while retaining security, and to permit network QoS. To ease adoption and

migration, AMQP is also designed to encompass JMS (Java Message Service) semantics. JMS is a hugely popular API for Java programmers and cannot be ignored. AMQP goes further, however, and includes additional semantics not found in JMS that members of the working group have found useful in delivering large, robust systems over the decades. Interestingly, AMQP does not itself specify the API a developer uses, though it is likely that will happen in the future.

An example feature not found in JMS is AMQP's Mandatory Delivery Mode, in which a client can use AMQP to request services from a pool of servers connected to AMQP broker queues. The AMQP broker can load-balance requests among the services subscribed to a request queue, and the number of processes providing a service can dynamically grow and shrink with no impact on the client(s). If the service pool shrinks to zero, however, the client can be informed by AMQP using the Mandatory Delivery Mode since that may be an operational error for the application.

AMQP also specifies a small wire-level type system for message properties, enabling them to be read efficiently by many programming languages, as well as by the MOM servers themselves for filtering and routing purposes. Thus, not only can a Python client read headers set in Java servers, but different vendors can relay messages between their implementations seamlessly. The type system, however, suffers the usual problem in that object

FIG 1

Toward a Commodity Enterprise Middleware

types cannot be transported in the headers; what would a Cobol program do with a Smalltalk object?

Finally, AMQP draws heavily on the heritage of IETF open standards. It tries not to reinvent existing concepts. Early versions of the AMQP wire protocol were influenced by SMTP,³ MIME,⁴ HTTP-NG,⁵ NFSv4,⁶ SCTP,⁷ BEEP,⁸ and the writings of Marshal Rose,⁹ an IETF veteran.

MAIN FEATURES

AMQP is split into two main areas: transport model and queuing model. AMQP is unusual in that it thoroughly specifies the semantics of the services it provides within the queuing model; since applications have a very intimate relationship with their middleware, this needs to be well defined or interoperability cannot be achieved. In this respect, AMQP semantics are more tightly defined than JMS semantics.

As stated, AMQP's transport is a binary protocol using network byte ordering. We wanted to make it easy to embed AMQP inside the ASICs (application-specific integrated circuits) of network elements, by design. With free tools such as Wireshark, it is not necessary to use XML for technical infrastructure layers that only specialists will see. XML has been used in the likes of BEEP and XMPP: in the case of BEEP it complicates the protocol; in the case of XMPP it is limited to being carried on a stream-oriented transport. AMQP aims to be high performance and flexible, to be hardware friendly rather than human friendly. The protocol *specification* itself, however, is written in XML so implementers can code-generate large portions of their implementations; this makes it easier for vendors to support the technology.

The transport model itself can reuse different underlying transports. The first is TCP/IP, but by adopting SCTP, we can obtain better parallelism for messages (SCTP removes the byte-stream head-of-line blocking problem imposed by TCP). There are also planned mappings to UDP to support AMQP over multicast, and bindings to InfiniBand are planned. InfiniBand's performance is generating a lot of interest at banks, and AMQP would make it very accessible to developers.

TCP/IP, however, is expected to be the default choice for most end users for best interoperability. With these options emerging, it is important for AMQP to establish a useful functional default set of capabilities that all implementations must adhere to or suffer the lowest-common-denominator problem that plagued protocols such as early versions of NFS (many servers did not implement file locking). Hopefully, a compliance testing kit will address this issue.

MESSAGES

Messages in AMQP are self-contained and long-lived, and their content is immutable and opaque. The content of messages is essentially unlimited in size; 4GB messages are supported just as easily as 4KB messages. Messages have headers that AMQP can read and use to help in routing.

You can liken this to a postal service: a message is the envelope, the headers are information written on the envelope and visible to the mail carrier, who may add various postmarks to the envelope to help deliver the message. The valuable content is within the envelope, hidden from and not modified by the carrier. The analogy holds quite well, except that it is possible for AMQP to make unlimited copies of the messages to deliver if required.

QUEUES

Queues are the core concept in AMQP. Every message *always* ends up in a queue, even if it is an in-memory private queue feeding a client directly. To extend the postal analogy, queues are mailboxes at the final destination or intermediate holding areas in the sorting office.

Queues can store messages in memory or on disk. They can search and reorder messages, and they may participate in transactions. The administrator can configure the service levels they expect from the queues with regard to latency, durability, availability, etc. These are all aspects of implementation and not defined by AMQP. This is one way commercial implementations can differentiate themselves while remaining AMQP-compliant and interoperable.

EXCHANGES

Exchanges are the delivery service for messages. In the postal analogy, exchanges provide sorting and delivery services. In the AMQP model, selecting a different carrier is how *different ways of delivering the message* are selected. The exchange used by a publish operation determines if the delivery will be direct or publish-and-subscribe, for

example. The exchange concept is how AMQP brings together and abstracts different middleware delivery models. It is also the main extension point in the protocol.

A client chooses the exchange used to deliver each message as it is published. The exchange looks at the information in the headers of a message and selects where they should be transferred to. This is how AMQP brings the various messaging idioms together—clients can select which exchange should route their messages.

Several exchanges must be supported by a compliant AMQP implementation:

- The direct exchange will queue a message directly at a single queue, choosing the queue on the basis of the “routing key” header in the message and matching it by name. This is how a letter carrier delivers a message to a postal address.
- The topic exchange will copy and queue the message to all clients that have expressed an interest based on a rapid pattern match with the routing key header. You can think of the routing key as an address, but it is a more abstract concept useful to several types of routing.
- The headers exchange will examine all the headers in a message, evaluating them against query predicates provided by interested clients using those predicates to select the final queues, copying the message as necessary.

Throughout this process, exchanges never store messages, but they do retain binding parameters supplied to them by the clients using them. These bindings are the arguments to the exchange routing functions that enable the selection of one or more queues.

BINDINGS

The arguments supplied to exchanges to enable the routing of messages are known as bindings (see figure 2). Bindings vary depending on the nature of the exchange; the direct exchange requires less binding information than the headers exchange. Notably, it is not always clear

which entity should provide the binding information for a particular messaging interaction. In the direct exchange, the sender is providing the association between a routing key and the desired destination queue. This is the origin of the “destination” addressing idiom so common to JMS and other queuing products.

In the topic exchange, it is the receiving client that provides the binding information, specifying that when the topic exchange sees a message that matches any given client(s’) binding(s), the message should be delivered to all of them.

AMQP has no concept of a “destination,” since it does not make sense for consumer-driven messaging. It would limit its abstract routing capabilities. The concept of bindings and the convention of using a routing key as the default addressing information overcome the artificial divisions that have existed in many messaging products.

IMPLEMENTATIONS

A standard is nothing without implementations. AMQP has several available now. The first implementation is iMatix OpenAMQ (<http://www.openamq.org>), which is a C implementation of the server in production use at JPMorgan.

Apache’s Qpid project has entered incubation (<http://incubator.apache.org/projects/qpid.html>). It will be Apache’s multilanguage implementation of AMQP, with servers available in C++ and Java, and clients available in C, C++, Java JMS, Python, and Ruby on Rails, with more to follow. Qpid has a very active development community and is making rapid progress toward its M2 release. Also, Qpid is being used as the basis for several commercial offerings, notably from IONA and Red Hat.

Most recently, and intriguingly, Rabbit Technologies has developed an implementation on the Erlang/OTP (Open Telecom Platform), building on that language’s strong support for parallelism and networking.

Of course, you should be able to mix and match client and server components from any vendor product—Qpid’s Java client talking to RabbitMQ’s Erlang server, for example.

The best way to learn more about AMQP is to visit <http://www.amqp.org>, where you can download the specification, try one of the free implementations,

Toward a Commodity Enterprise Middleware

or even build your own. Both the protocol working group and the implementation groups are open to review and feedback; we want this protocol to be useful, successful, and inclusive.

A TRANSPORT FOR OTHER STANDARDS

In addition to being used to support organizations' internal messaging needs, AMQP is useful as a standard transport for other standards; after all, open standards should be built using other open standards.

Many business messaging standards describe business transactions, but there are only a relatively small number of transport options. HTTP is often used simply because it is perceived to be generally available, and firewalls are often configured to let HTTP traffic pass. HTTP offers only basic push messaging semantics, however; it does not enable any kind of call-back or event notification, and it requires the applications that use it to build their own message store, reliability, and access control facilities on top of it. EDI AS2 and WS-RM are examples of standards layered on HTTP; but these describe only message transport, not the semantics of how a message is delivered to an application. This leaves the developer with a partial solution and more problems to solve.

By offering AMQP as a standard for business messaging, we can make the full richness of messaging middleware available to other standards. Using AMQP servers, we can remove the burden of providing availability and reliability from end-user applications, making them simpler, cheaper, and more functional. The only requirement is that an AMQP server is run and that TCP/IP port 5672 be opened in the firewall. Given that any commercial activity involves lawyers and contracts, opening a port is a small price to pay to gain rich messaging functionality.

The TWIST standards organization, which promotes standards around financial supply chain management, is a member of the AMQP working group and promotes AMQP as its preferred standard for transporting business messages.

The FIX community is also looking at AMQP as a possible transport layer for FIX 5. FIX is a popular trad-

ing protocol that has recently been extended to support efficient delivery of market data; the current session layer, however, cannot provide publish-subscribe or scalable guaranteed delivery. AMQP offers FIX users the chance to get the features they need, while remaining open.

SOA WITH AMQP

SOA (service-oriented architecture) is a technique for building large systems out of highly cohesive services loosely coupled to provide business processes. SOA is not a new concept. It was well known in the mid-'90s and earlier, but in its latest incarnation it is being pitched as *Web service-oriented architecture*. The architectural pattern needs neither HTTP nor XML.

One of the most fundamental parts of SOA is the communications mechanism that links the services. The term EMB (enterprise message bus) is closely linked to SOA.

Traditional deployments of EMBs have used proprietary technology, but enterprises would rather have standards-based solutions that are open, as well as the ability to choose between and switch suppliers, and the improvements that competition brings. AMQP, therefore, represents an ideal choice for an EMB. It allows a clean migration away from proprietary protocols and provides an avenue into other standards, including Web services.

Web services has four basic parts: service description, XML message content, service discovery, and transport. The transport is commonly presumed to be HTTP, but it does not have to be. Enterprises often use XML over messaging middleware as the transport for all the benefits that brings. Having done this, enterprises find they have created the problem they wanted to avoid: running an open architecture over a proprietary transport. Combining Web services with AMQP as a transport gives the richness an enterprise needs with the openness it craves in its core architecture.

Use of AMQP in SOA is just beginning, and you don't need anything other than AMQP to do it. I am already involved in a project to migrate an existing mission-critical EMB from a proprietary middleware to AMQP, so we can cost-effectively scale the bus to many more systems.

CONNECTING TO LEGACY MIDDLEWARE

AMQP is a complete middleware protocol. It is not a lowest-common-denominator solution, and the only political design constraint is its explicit support for JMS semantics. Obviously, software that implements the AMQP specification will be able to interoperate, even where that software is from different suppliers. That means that the JMS client from product A would be able to talk to the C++ server

from product B and send messages to a Ruby client in product C. Of course, in the real world there will be some teething difficulties, and the AMQP working group is starting to focus on how to create and manage a protocol compliance test suite to mitigate that risk.

There are many middleware products, however, each with its internal architecture. The AMQP Working Group encourages middleware vendors to implement AMQP in their products, but it will be nontrivial to get a good semantic match, and some vendors may be reluctant to support interoperability for commercial reasons.

In the meantime, there will be a need for bridging from the installed base of proprietary products. The easiest way to do this may be to use one of the many commercial or open source EAI (enterprise application integration) packages, but we expect several AMQP products to include native bridges to the most common proprietary middleware soon.

ADOPTING AMQP

The most natural way for an organization to adopt AMQP is to deploy it opportunistically or as part of a strategy to move to standards-based EMB. This approach can enable an organization to benefit from competitively priced or open source solutions and gain experience with the protocol and products that support it. This is the approach we took: using AMQP in isolated systems and then branching into core EMB systems.

Over the course of a few years, much of your messaging may become AMQP-enabled by following this approach. Your current middleware supplier may adopt AMQP and enable your move to a standards-based model in that way. It is likely that AMQP will eventually be provided as a core service by your network infrastructure, in hardware.

On the other hand, if your company is embarking on SOA (or other bus-based architecture), we suggest that you seriously consider an en-masse deployment of AMQP as the backbone of SOA. Doing so may position you to benefit from competition among suppliers of AMQP-compatible middleware so you may achieve the levels of support and qualities of service you need from a suitable supplier. Deploying AMQP may also mitigate the worrisome issue of vendor lock-in or supplier failure for the critical bus component of an SOA, which is a long-term investment for your company. This same thinking applies where open protocols for e-business are being deployed between trading partners over the Internet or leased lines. Of course, whatever strategy you choose is yours alone and must be determined by your circumstances.

CONCLUSION

After two decades of proprietary messaging middleware, a credible standards-based alternative exists at last. The AMQP Working Group is rapidly evolving the protocol and hopes to reach version 1.0 during 2008, but implementations are available today that are both useful and proven in real-world deployments.

AMQP lets more applications be enterprise-grade without the costs associated with that label. It provides a capable messaging backbone that can be a springboard for new innovations and new kinds of applications. Q

REFERENCES

1. O'Hara, R.J., Hintjens, P., Greig, R., Sim, G., et al. 2006. AMQP Advanced Message Queuing Protocol Version 0.8; <http://www.amqp.org/tikiwiki/tiki-index.php?page=Download>.
2. Bradner, S. 1996. IETF Standards Process, RFC 2026.
3. Postel, J., Klensin, J. 1982/2001. Simple Message Transfer Protocol, RFC 821 / 2821.
4. Freed, N., Borenstein, N. 1996. Multipurpose Internet Mail Extensions (MIME), RFC 2045.
5. Janssen, B. 1998. Binary Wire Protocol for HTTP-ng; <http://www.w3.org/Protocols/HTTP-NG/>.
6. Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., Noveck, D. 2003. Network File System (NFS) version 4 Protocol, RFC 3530.
7. Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., Paxson, V. 2000. Stream Control Transmission Protocol, RFC 2960.
8. Rose, M. 2001. The Blocks Extensible Exchange Protocol Core, RFC 3080.
9. Rose, M. 2002. "Introduction: Application Protocol Design." *BEEP: The Definitive Guide*. O'Reilly.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

R. JOHN O'HARA is a senior architect and distinguished engineer at JPMorgan. He drove the creation of AMQP and was an early contributor to the FpML financial markets messaging standard. He is chairman of the AMQP Working Group. His interests include data center virtualization, creative uses of open source for commercial data processing, event and data replication middleware, and making software architecture "simple and intuitive." O'Hara holds an honors degree in electronic engineering and computer science from Aston University, England.

© 2007 ACM 1542-7730/07/0500 \$5.00