AMQP Essentials

BY PAOLO PATIERNO

WHAT IS AMQP?

AMQP (Advanced Message Queuing Protocol) is a binary transfer protocol that was made for enterprise applications and server-to-server communication (e.g., for financial businesses), but today it can be very useful in the Internet of Things world, thanks to the following primary features. AMQP is **binary** and avoids a lot of the useless data sent on the wire when using a text-based protocol like HTTP; because of this, it can be considered **compact**, too. Thanks to its **multiplexed** nature, only one connection (over a reliable stream transport protocol) is needed to allow separated data flows between the two peers; and of course it's **symmetric** and provides both a client-server communication style and peer-to-peer exchange. Finally, it's **secure** and **reliable**, providing three different levels of QoS (Quality of Service).

The last ratified version of AMQP (1.0) is the only one standardized by OASIS (since 2012/10) and ISO/IEC (since 2014/05), and it's totally broker-model agnostic, as it doesn't define any requirements on broker internals (this is the main difference with previous "not standard" versions like 0.9.1); the protocol is focused on how the data is transferred on the wire.

AMQP ARCHITECTURE

AMQP has a layered model defined in the following way from a bottom-up perspective:

Transport/Framing: Defines the connection behavior and the security layer between peers on top of an underlying network transport protocol (TCP, for example). It also adds the framing protocol and how the exchanged data is formatted and encoded.

- **MESSAGING:** Provides messaging capabilities at application level on top of the previous layer defining the message entity as built of one or more frames.

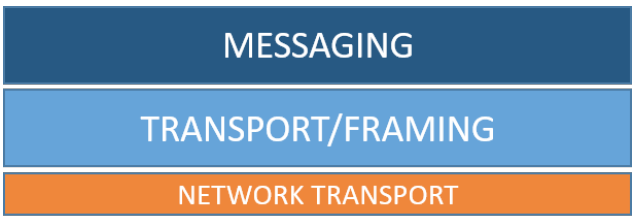

FIGURE 1: *Layered model*

Regarding the network transport layer, AMQP isn't strongly tied to TCP, and as such can be used with any reliable stream transport protocol; so, for example, SCTP (Stream Control Transmission Protocol) and pipes are suitable.

TRANSPORT/FRAMING

The main entities that build peers in an AMQP network are:

- **NODES:** Named entities responsible for storing and/or delivering messages. In the messaging space, a node could be, for example, a producer/consumer or a queue. A node is addressable and can be organized in a flat, hierarchical, or graphical way.
- **CONTAINER:** Generally speaking, a container is an application. Previously defined nodes live in a container that could be a client with its producers and/or consumers, or a broker with its storage entities (like queues for example)

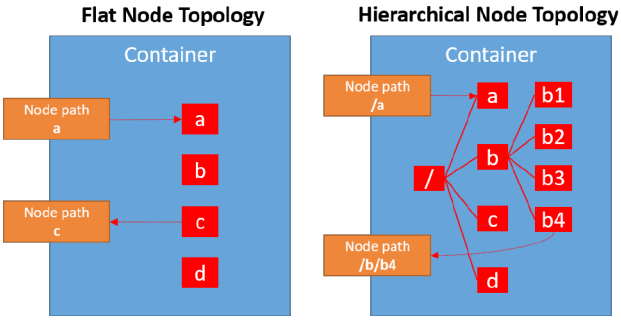

FIGURE 2: *Nodes and Containers*

So, at the transport level, how do containers and related nodes communicate with each other and exchange messages? First of all, a TCP connection is established between the containers with or without a security layer using SSL/TLS protocol. After that, an AMQP connection is created on top of the underlying network connection thanks to the exchange of some connection preamble packets with protocol version information; the same transport-level security (like, for example, SSL/TLS) can be negotiated in line using SASL (Simple Authentication and Security Layer) protocol. Such a connection provides a full-duplex communication with an ordered sequence of frames whose maximum size is negotiated to provide a first level of flow control. The same connection is divided in multiplexed and unidirectional channels, and all the frames flow through them with an assigned channel identifier.

After the connection, an AMQP session is established between two peers; it binds two unidirectional channels to form a bidirectional conversation with a flow control mechanism based on the number of exchanged frames. Of course, a connection supports multiple sessions.

Finally, in order to exchange messages between nodes (for example, from a producer to a queue and from the queue to a consumer) an AMQP link is created between them. It's a unidirectional route attached to each node at a terminus that could be the source or target, and it's responsible for tracking a message's exchange status. The link provides the third level of flow control based on credits. Of course, a session can support multiple links.

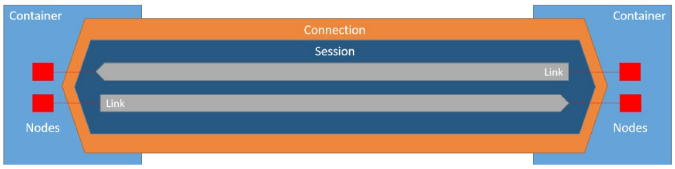

FIGURE 3: *Connection, sessions, and links*

Connections and sessions are ephemeral, so they don't retain any status if the underlying network connection is lost; on reconnecting, the peer has to create a new connection and a new session not related to the previous ones. Links, however, are recoverable, and—if the network goes down during message transfer—each link is recovered with the previous message delivery status (related to the QoS requested).

Regarding the data exchanged, the frame is the atomic unit and is divided in three main parts:

- **FRAME HEADER:** The header has a fixed size (8 bytes), and it is mandatory, as it contains the information needed to parse the rest of the frame itself—for example the total frame size and the frame type.
- **EXTENDED HEADER:** A variable header that depends on the frame type.
- **FRAME BODY:** A sequence of bytes that has a format that depends on the frame type.

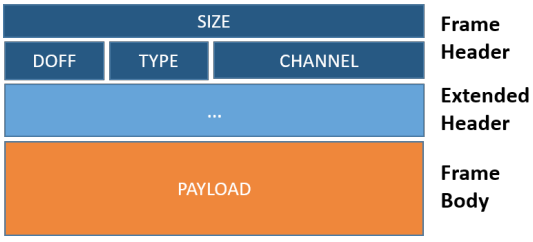

FIGURE 4: *Frame format*

The frame body is interesting because it's defined as a performative followed by an opaque payload filled by the application with data to transmit; these performatives are related to opening/closing the connection, beginning/ending a session, attaching/detaching a link, transferring content, and handling flow control.

MESSAGING

The applications based on the AMQP protocol doesn't exchange data speaking the framing "language," but rather it's the messaging layer built on top of it that provides messaging capabilities.

This layer defines a well-known structure of the message composed of two main parts :

- **BARE MESSAGE:** it's an immutable part from the sender to the receiver. No one intermediary can change its content
- **ANNOTATED MESSAGE:** it consists of the previous bare message plus some annotations that can be used and changed by intermediaries between sender and receiver

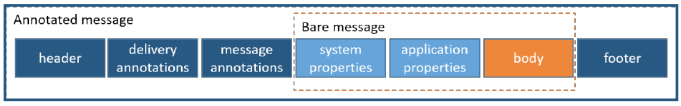

FIGURE 5: *Message format*

The bare message contains the body and two types of collections: the first one is for system properties that are standard and well-defined by the AMQP specification; the second one is for application specific properties (also named user properties) that can be added and changed by the application.

The annotations are flexible and related to the overall solution in terms of clients and broker for example; the same is for application properties that aren't fixed but created by the application according to its needs. The system properties are well-known and defined by the specification and most used are:

- **MESSAGE ID:** A unique identifier for the message assigned by the producer.
- **TO:** Identifies the destination node for the message.
- **SUBJECT:** Contains summary information about message content.
- **REPLY TO:** The address of the node to send replies to (it's useful in a request/response scenario).
- **CORRELATION ID:** Used for correlation between a request message and related response (in a reply message it's the message id of the request).
- **CONTENT TYPE:** Used to specify content type for the opaque payload.
- **ABSOLUTE EXPIRY TIME:** The time when the message is considered to be expired.

The messaging layer defines a set of delivery states to describe the message state at the receiver endpoint. A delivery state can be "terminal," which means that the message won't change anymore and results in what is called an "outcome." A message can also be "non-terminal," which indicates a transient state used for link recovery.

The outcomes defined by the specification are:

- **ACCEPTED:** The message is received and successfully processed by the receiver.
- **REJECTED:** The receiver rejected the message because it is invalid and can not be processed.
- **RELEASED:** The message was not processed, even if it was a valid message (i.e. not rejected). It should be redelivered.
- **MODIFIED:** indicates that the message was modified but not processed

There is only one "non-terminal" state, named **Received**, which indicates partial message data or the starting point for a resumed transfer.

AMQP COMMUNICATIONS

All the AMQP concepts—from connection, session, and link to performatives and messages—fit together to define how the communication happens between two peers. The main steps involved are:

- **OPEN/CLOSE** a connection (respectively after opening a network connection and before closing it) using "open" and "close" performatives
- **BEGIN/END** a session inside the connection thanks to "begin" and "end" performatives
- **ATTACH/DETACH** a link inside the session using "attach" and "detach" performatives
- **SEND/RECEIVE** messages with flow control thanks to "transfer," "disposition," and "flow" performatives

COMMUNICATION: OPEN

In order to open the communication with a peer, first there is an AMQP/SASL handshake on the raw TCP connection, then the AMQP "open" performative is exchanged to define the max frame size (flow control), maximum number of channels, and so on. Inside the connection, a session is started using the "begin" performative specifying the window size (number of frames for flow control). Finally, the "attach" performative is used to attach a link.

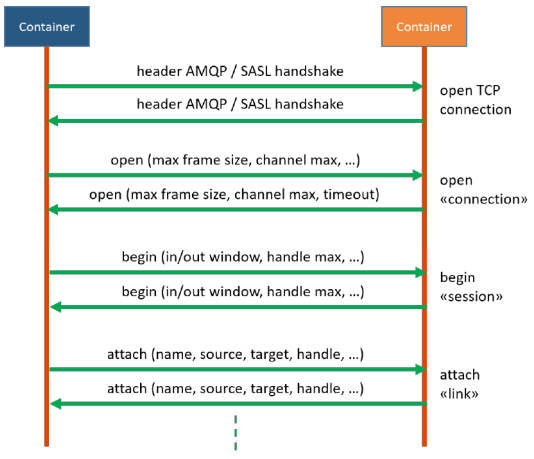

FIGURE 6: *Communication: Open*

COMMUNICATION: SEND

After the link is attached, the receiver can send a "flow" performative to the sender specifying the credit number to limit number of messages it's able to receive (flow control). The producer sends data using the "transfer" performative, which is followed by a "disposition" performative by the receiver **if and only if** the required QoS is at level one (at least once) and the messages are not settled by the producer. The receiver can send **only one** "disposition" performative to confirm that it has received more "transfer" performatives.

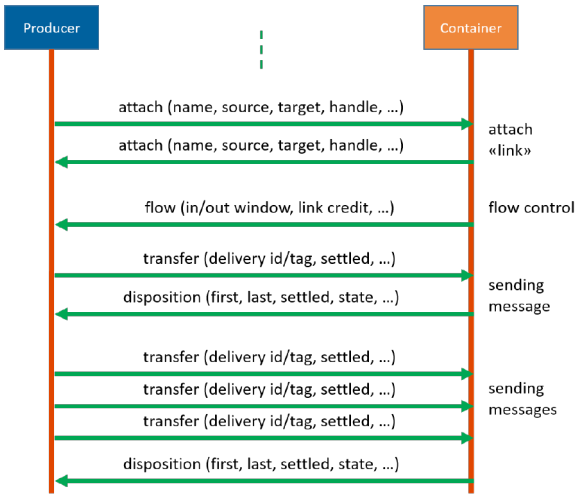

FIGURE 7: *Communication: Send*

COMMUNICATION: RECEIVE

A receive communication is the opposite flow of a send. The receiver sends the "flow" performative to set the credit-based flow control and how many messages it can receive before processing them. For one or more "transfers," it replies with a "disposition" if the QoS level is greater than zero and the producer requires settlement.

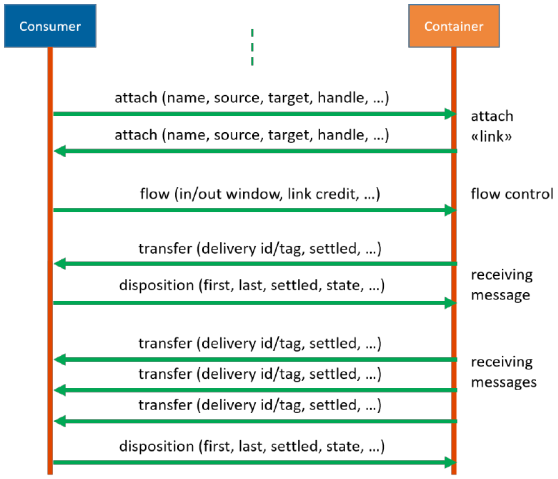

FIGURE 8: *Communication: Receive*

COMMUNICATION: CLOSE

Closing a communication means it becomes detached from all active links using the "detach" performative. After that, the "end" performative is used to end the session, and the "close"

performative is used to close the connection. Of course, the last step is to close the underlying network connection at the socket level.

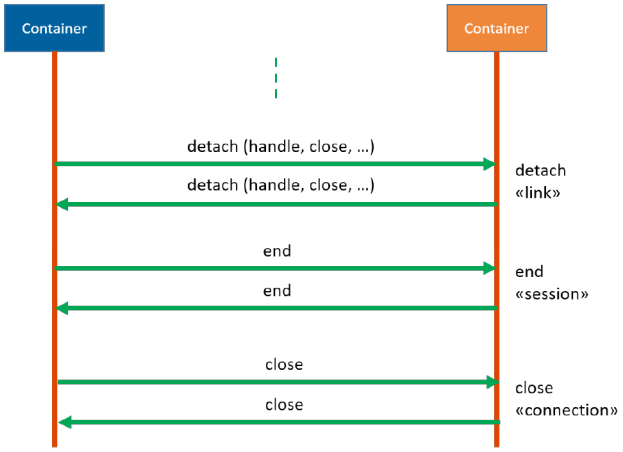

FIGURE 9: *Communication: Close*

FLOW CONTROL

The AMQP protocol provides different ways to do "flow control" at different levels. First of all, during the opening connection process, the two peers negotiate a maximum frame size that defines the maximum size of each single frame that can be exchanged.

The first level of flow control is provided by session; each session endpoint has an incoming and outgoing window with a size defined as frame count. On both sides (sender and receiver), the frame's exchange can be stopped when the window is full (the sender doesn't have more window space to send, the receiver doesn't have more window space to receive); each transfer decrements the window size.

The last flow control level is at the messaging level, where each link has a "link credit," which is the number of messages the receiver is able to receive. The receiver can set this value using the "flow" performative; for each incoming message the counter is decremented until a value of zero suspends transfer.

Why are two different flow control levels needed?

The session flow control is useful to protect a Cloud platform at high-scale when there are million connections and frames; in this way it can handle capacity management and throttling. In the IoT space it's very useful for very low constrained devices that have low memory capacity for buffers.

The link flow control is at higher level and protects the application in order to avoid the need to accept more than messages that the application can handle concurrently.

QUALITY OF SERVICE

The AMQP protocol provides the three following Quality of Service levels related to the messages delivery:

- **AT MOST ONCE:** it's also known as "fire and forget," because the message can be delivered at most one time. It could happen that it's lost in the network and doesn't arrive to the receiver. The sender doesn't receive any information about message receipt and doesn't resend the message.
- **AT LEAST ONCE:** in this scenario the message can be delivered one or more time. For each message, the sender should receive an acknowledge by the receiver. If the receive gets the message but the acknowledge is lost then the sender re-sends the message (in that case a second delivery as duplicate message).
- **EXACTLY ONCE: BEGIN/EN** thanks a kind of double commit with some acknowledge messages exchanged between sender and receiver, the message is delivered exactly only one time to the receiver.

When the message exchange starts, the sender assigns a delivery tag to the message in order to track its delivery. Both peers have an internal map with settlement status of the messages in transit and each message starts with an unsettled status. The transfer performative is used to send the message, and it contains the settlement status at sender too; the disposition performative is used as acknowledge to describe the settlement status at receiver.

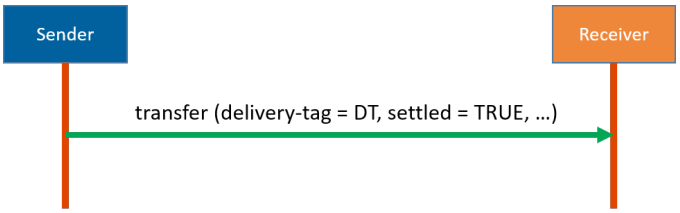

FIGURE 10: *At most once delivery*

In the "at most once" delivery, the sender sends the message already settled (settled = TRUE); it doesn't want to know about delivery anymore. The message could be arrived or not but no acknowledge is expected.

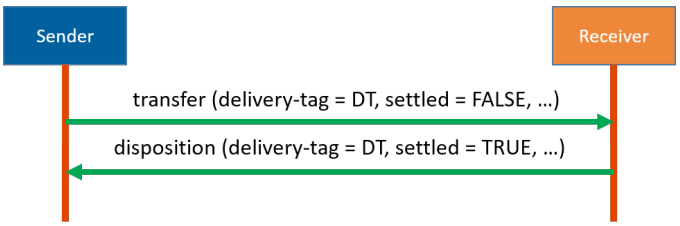

FIGURE 11: *At least once delivery*

When the "at least once" delivery is used, the sender sends the message in unsettled status (settled = FALSE) because it wants an acknowledge from the receiver; during the transit, the message is still unsettled in the internal sender map. Once received the message, the receiver replies with a disposition frame that defines the message as settled (at receiver side of course); receiving the disposition, the sender changes settlement status to settled and the message isn't in transit anymore. Of course, the disposition frame could be lost and the sender re-sends the message having a duplicate delivery to the receiver.

The "exactly once" delivery isn't commonly used.

SECURITY AND AUTHENTICATION

SSL/TLS AND SASL

The AMQP protocol provides security features at connection level and it's based on the SSL/TLS protocol other than using SASL (Simple Authentication and Security Layer) authentication mechanism.

Using SSL/TLS the communication channel is encrypted to provide "confidentiality" so that an eavesdropper can't get and understand data in transit between the peers. Establishing the SSL/TLS connection can be done in the following three different ways:

- **TLS INSIDE AMQP:** the connection between peers starts at AMQP level directly (on port 5672) and then the TLS handshake occurs inline. During the AMQP negotiation, the peer indicates desire for TLS inside.
- **AMQP INSIDE TLS:** the peers starts a TLS handshake first on top of the underlying TCP protocol. In that way the network layer is encrypted and then the AMQP negotiation can start. In this scenario, the different port 5671 is used (it's related to AMQPS, so AMQP on SSL/TLS).
- **WEBSOCKET TUNNEL:** even if it's still in draft, there is a specification about AMQP 1.0 over WS. In this case, a WS channel is established inside an already TLC encrypted connection and then the AMQP handshake starts on top of it.

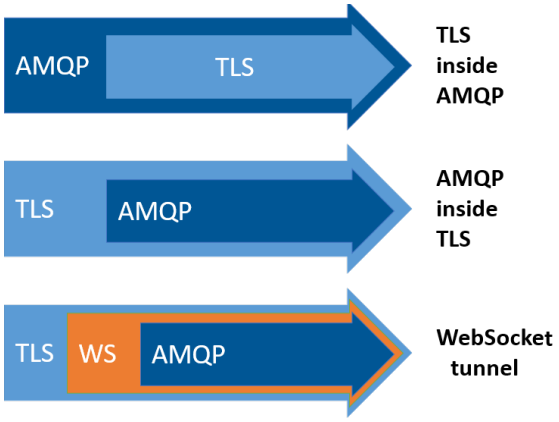

FIGURE 12: *AMQP & TLS*

Other than security, AMQP provides a SASL based authentication mechanism. The type of authentication to use is negotiated during a SASL handshake with specific AMQP frames and it could be one of the main supported like ANONYMOUS (no authentication), PLAIN (username and password), EXTERNAL and so on.

PLATFORMS AND CLIENTS BASED ON AMQP

Who uses AMQP? What are the main platforms based on AMQP?

The first offer comes from the open-source world thanks to the Apache Foundation with the Apache Qpid project that provides an AMQP stack implementation in C, Java, C++ and other languages. In the same way, there is the AMQP .Net Lite library that is an open source .Net and C# implementation from Microsoft.

The above products are used as clients in most cases but what about broker/server and enterprise architecture based on AMQP?

ActiveMQ by Apache Foundation is the most popular and powerful open-source messaging and integration patterns server that support AMQP protocol.

There is the Microsoft Azure Service Bus that provides a messaging middleware exposing queues and topics/subscriptions with which we can communicate using AMQP. This service offers the event hubs too for big data ingestion. Another service by Microsoft based on AMQP is the Azure IoT Hub for Internet of Things solution and devices connection.

After that, another enterprise product and messaging middleware is JBoss A-MQ by Red Hat that is built upon the ActiveMQ project.

Last but not the least and always as messaging middleware, there is MQLight by IBM that added AMQP to the already supported MQTT protocol.

Other products are available on the official AMQP website here.

LET'S TRY AMQP

The easiest way to start with an AMQP protocol client-side would be to use a high level language implementation like Java (Apache Qpid Proton-J) or C# (AMQP .Net Lite). On the broker side, to avoid using a Cloud platform, an ActiveMQ instance running locally on the PC is a great choice.

The following example is a simple message exchange using a queue between two clients, a sender and a receiver; the queue is an AMQP node created inside the ActiveMQ broker as container.

The ActiveMQ broker is available here, and the getting started guide to install it and create a queue (using the Web UI console) is here.

The AMQP .Net Lite library is open source, and you can clone the code from GitHub.

OPEN COMMUNICATION

First of all we need to define the base address of the broker for connecting. For this purpose, the AMQP .Net Lite library exposes the Address class that we can use in the following way:

```
Address address
        = new Address("amqp://admin:admin@192.168.1.103:5672");
```

That has the format `amqp://[username]:[password]@[host]:[port]`.

Opening the communication with the broker means open a connection and begin a session:

```
Connection connection = new Connection(address);
Session session = new Session(connection);
```

After that we need to create and attach the link with the destination queue to send data:

```
SenderLink sender
        = new SenderLink(session, "sender-link", "q1");
```

In this example, the queue already created and available in the broker is named "q1".

At the end of this three steps, the connection is established and the sender can start to send messages to the queue.

SEND MESSAGE

Each message is implemented through the Message class that exposes system and application properties other than a body that can be filled with any payload.

```
Message message = new Message("Hello DZone!");
message.Properties = new Properties();
message.Properties.MessageId = messageId;
message.ApplicationProperties = new ApplicationProperties();
message.ApplicationProperties["my_app_prop"] = value;
sender.Send(message, 60000);
```

After creating the message, the `Send()` method provided by the SenderLink class is able to send the message in a synchronous way with a specified timeout.

RECEIVE MESSAGE

On the receiver side, the connection and the session are created in the same way but the attached link is specified using the `ReceiverLink` class.

```
ReceiverLink receiver
        = new ReceiverLink(session, "receiver-link", "q1");
```

This class provides the `Receive()` method to wait synchronously for an available message in the queue and get it. The returned message can be "null" if the receiving timeout expires.

```
Message message = receiver.Receive();
```

After receiving the message, the receiver needs to update the delivery status calling the `Accept()`, `Reject()` or `Release()` method with message as parameter to complete the action.

```
receiver.Accept(message);
```

Before receiving the message, the receiver can use the `SetCredit()` to apply flow control at link level to specify the maximum number of messages it can handle.

```
receiver.SetCredit(10, false);
```

RECEIVE MESSAGE ASYNCHRONOUSLY

The AMQP .Net Lite provides an asynchronous way to receive messages using a callback invoked when a message is received from the queue.

```
receiver.Start(10, (link, m) =>
    {
        // do work with message
        link.Accept(m);
    });
```

In that case, the `Start()` method is used to specify both the credit and the lambda expression used as callback that has the link and the received message as parameters.

CLOSE COMMUNICATION

When the peers don't need the channel anymore, they can close it in the opposite order they open it. First detaching the link then ending the session and finally closing the connection.

```
sender.Close(); // or receiver.Close();
session.Close();
connection.Close();
```

ABOUT THE AUTHOR

PAOLO PATIERNO is a Senior Software Engineer and Microsoft MVP for Windows Embedded / Internet of Things who has been working on Microsoft technologies since 2006 with all .Net Frameworks (Micro, Compact and Full); he has been developing on embedded and mobile systems (based on Windows CE, WindowsPhone/Android and RTOS) since 2010, using C/C++, C# and Java. Focused on IoT protocols and on developing end to end solution in the IoT business on both devices and Cloud side. Member of DotNetCampania, TinyCLR.it and Embedded101 communities. Technical writer and owner of some open source projects on GitHub. **He blogs at paolopatierno.wordpress.com and you can follow him on Twitter @ppatierno.**

RESOURCES

AMQP 1.0 SPECIFICATION
docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

AMQP.ORG : OFFICIAL WEB SITE
amqp.org

AMQP VIDEO COURSE (by Clemens Vasters)
channel9.msdn.com/Blogs/Subscribe/The-AMQP-10-Protocol-16-Overview

BLOGS AND OTHER RESOURCES (from Apache Qpid)
qpid.apache.org/resources.html

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

VERSION 1.0