predic8

ActiveMQ, Qpid, HornetQ and RabbitMQ in Comparison

By: Thomas Bayer
Date: 04/11/2013

Newer architectures and the standardized AMQP protocol have led to a flood of message brokers. All brokers take claim to be fast, robust and reliable. But what really distinguish the broker? How do I choose the right broker? Should we continue to use established brokers such as the ActiveMQ or try a more modern one? This article attempts to answer these questions and help the reader in selecting a suitable broker.

The broker described in this article needed to:

- be available under an open source license
- allow access from Java
- offer *Quality of Service* features such as guaranteed delivery and persistence

In the next sections the broker will be presented with their specialties. Then we handle criteria such as persistence, supported platforms, performance and distribution. The description starts with the established ActiveMQ broker.

Apache ActiveMQ

Apache ActiveMQ has the largest number of installations of all open source message broker with the largest distribution. ActiveMQ implements the *Java Message Service* specification and offers numerous features such as support for the *Enterprise Integration Patterns*, for Spring Framework and for transactions.

For persistence of messages, different stores are used. The file-based database Kaha offers very good performance because it runs at Java VM of the broker and requires very few resources and has optimized for messages. has become popular too in enterprise environments. But the use of a relational database such as IBM's DB/2 or Oracle database. Using a JDBC store together with a relational database is associated with losing speed, but brings advantages for the operation: the database for persistence of messages can be incorporated into the operating of a data center and the existing mechanism can be used for the backup. It does not always make sense to create a backup for a message store.

A highlight at ActiveMQ is the numerous possibilities of clustering and distribution. In order to keep the operations going in case of failure of a broker ActiveMQ can be be operated in a *Master/Slave* configuration. Further, several brokers can be connected to a Network of Brokers. In a Network of Brokers queues and topics (publish / subscribe) are virtual, i.e. a queue is available on all nodes and the routing takes place automatically.

Figure 1 outlines a virtual broker which could be realized with distributed ActiveMQ installations.

Figure1: Virtual Broker with globally distributed nodes

ActiveMQ is used as bus infrastructure in some *Enterprise Service Bus* products such as ServiceMix and Mule ESB. There are many other projects that use ActiveMQ for transporting messages. Apache Geronimo and Tomee, the Java Enterprise Edition of the Tomcat Web container, are using ActiveMQ as JMS implementation.

The Fuse MQ Enterprise Broker by Progress Software is based on Apache ActiveMQ broker. Bugfix versions of Fuse are released more frequently as the versions of Apache ActiveMQ.

Since its inception in 2004, the ActiveMQ broker has matured and become widespread. Childhood diseases such as memory leaks have been eliminated long ago. For most tasks ActiveMQ messaging is reliable, high-performance and easy on resources.

ActiveMQ is still a very good choice for *Enterprise Messaging*. But ActiveMQ has gotten some competition, which is arriving with newer architectures, better performance, and with the support of standardized protocols in the area of the space deer.

Apache Apollo

Apache ActiveMQ Apollo gets competition from its own company. Apollo is a recent development based on the experience of the ActiveMQ project.

In order to be faster, more robust and easier to maintain than ActiveMQ, a completely new architecture was introduced. That architecture is based on Scala programming language, which supports well the development of concurrent systems. The threading of the Apollo broker differs fundamentally from that of the ActiveMQ. All tasks are performed asynchronously and non-blocking which contributes to increased performance and stability. This means that the skills of multi-core processors are better used.

For Apollo, there are two storages for persistent messages. On the one hand on the NoSQL database LevelDB by Google and on the other hand via the Java Edition of the Berkeley DB.

Through the HTML5 WebSocket Connector, a messaging client can be integrated directly into a web page. The following listing shows how to use JavaScript and the Apollo *stomp.js* library to access a broker from a web page. When connecting, a callback function is registered which displays the contents of a receiving message in a dialog.

```
1  var client = Stomp.client("ws://apollo.predic8.com:61623");
2
3  client.connect("admin", "pwd", function(frame) {
4     client.subscribe("/topic/like", function(msg) {
5        alert(msg.body);
6     });
7  });
```

Listing 1: Access to the Apollo broker with STOMP and JavaScript

Sending messages from within the browser can be done with following JavaScript one-liner:

```
1  client.send("/topic/like", {}, inhalt);
```

Although Apollo is written in Scala, the broker can be used safely in a Java environment. The fact that Apollo is realized with Scala can only be seen because the distribution also contains some Scala libraries among many Java libraries.

For the administration Apollo offers a simple web console as the following screenshot shows:

Figure2: Admin console of the Apollo Broker

If you edit the file extension of the URL from *.html* to *.json* you'll get a JSON representation rather than a Web page.

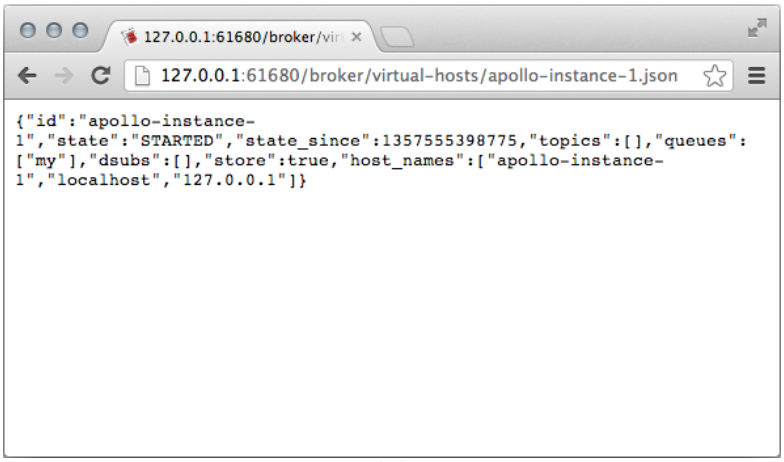

Figure3: Apollo console as JSON representation

By using REST and JSON you can also achieve write access to the Apollo Web application, for example to create a new queue. Figure 4 shows a screenshot with a visualization of the Apollo REST API. The visualization is integrated in Apollo and realized by the Swagger tool. [The description with Swagger] is a specification and a framework for the description of REST APIs.

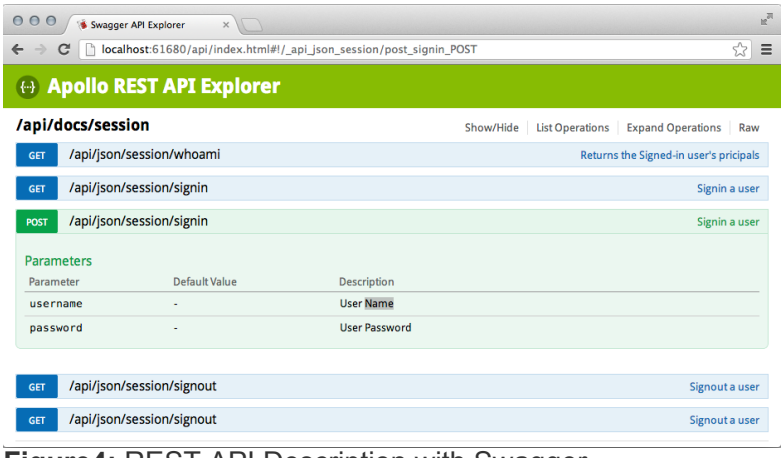

Figure4: REST API Description with Swagger

There is no client library for Apollo itself. Therefore other clients can be used for the protocols supported by Apollo: MQTT, OpenWire and STOMP.

FFMQ

The *Full-Java, native JMS, message Queuer* is a lightweight JMS implementation. The whole server is smaller than 600 KB but there are a number of restrictions regarding JMS conformity or transactions. If you can dispense with advanced JMS functionality such as *Message Groups* gets a simple, fast and straightforward message broker with FFMQ.

HornetQ

For a long time the basis code of HornetQ was known under the name of *JBoss Messaging 2.0* until it was a further developed in separate project under the name of HornetQ. HornetQ can be used independently of the JBoss Application Server. Since version 6 HornetQ is the preconfigured message broker for JMS of the JBoss server.

HornetQ is configured as Enterprise Message Broker. It has lots of features and lots of settings and is a complete implementation of the *Java Message Service* specification.

The User Manual has got 334 pages and therefore it is the most extensive of the listed Message Broker. The very detailed documentation contains the concepts of the architecture through to XA transactions and clustering.

HornetQ contains no stand-alone console for administration. HornetQ can be managed over the console of the JBoss Application Server. For example you can create queues or read statistics using JMX and MBeans. Since I would like to use the HornetQ in the standalone version a special bracket is on my wish list.

HornetQ has its strengths when used as Enterprise Broker and as JMS implementation. The integration into JBoss will certainly also help to its spread. This implies a certain degree of complexity. So there are more suitable broker if you need just easier configurations. For Enterprise applications HornetQ definitely has a sharp sting.

JBoss Messaging

The development of JBoss Messaging is set. New features will be implemented only in the follow-up project HornetQ. For a while there will be further support and bug fixes.

OpenJMS

OpenJMS was the open source JMS implementation from Sun Microsystems. In recent years, I could not see much activity around OpenJMS. It seems that this project is not especially alive.

Apache Qpid™

Besides ActiveMQ and Apollo there is another Apache Message Broker, Apache Qpid. The aim of the Qpid project is the 100 percent compatibility with the *Advanced Message Queuing Protocol Standard*.

Qpid broker is available for C++ and Java. This article describes the features of the Java version. For Java clients there is a JMS API for Qpid. For C++, Python and Microsoft's .NET there is the *Qpid Messaging API*.

For the persistence of the messages the relational Apache Derby database and the Oracle Berkeley DB are supported.

Based on Apache Qpid Red Hat offers the Enterprise Messaging product MRG. For MRG longtime support and versions with additional bug fixes are available.

Proton™, a subproject of Qpid, is a lightweight implementation of the AMQP protocol. Using proton it is likewise possible to develop clients and servers. Proton is available for C and Java. The C implementation also includes bindings for PHP, Python and Ruby.

RabbitMQ

The RabbitMQ broker was created by the functional language Erlang. Erlang is especially suited for distributed applications, as concurrency and availability is well-supported.

Do not be deterred from that RabbitMQ is implemented in Erlang. Its installation runs quickly and easily at Windows and Mac OS. For programming in Java or other languages client libraries are available.

Using a Plugin allows RabbitMQ to understand protocols such as STOMP. The core of RabbitMQ is fully geared to the AMQP protocol. Unfortunately, there is currently no support for AMQP in version 1.0. But support for AMQP 1.0 is already in the planning.

The management plugin provides an appealing web console that allows easy administration. There also are visualized statistics such as the number of messages per second and the consumption of resources, such as memory, sockets, and the crucial file descriptors.

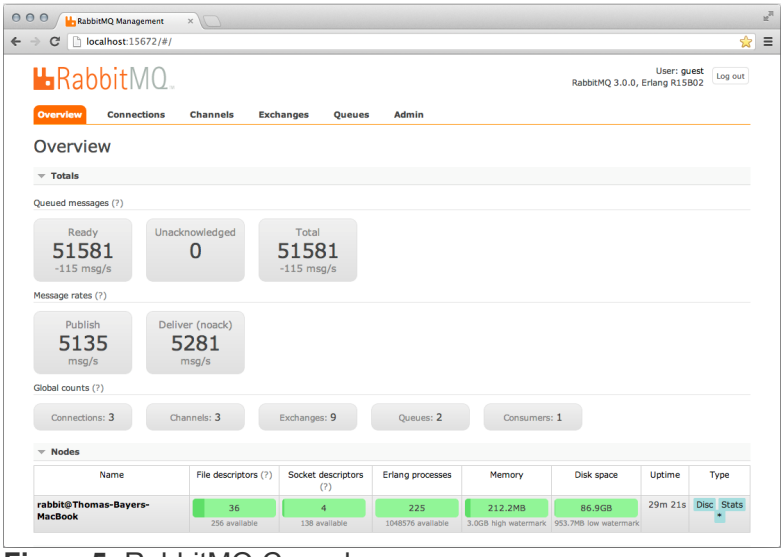

Figure5: RabbitMQ Console

ZeroMQ

ZeroMQ is no classic broker, which provides message queues to its clients, but rather a library to create distributed and concurrent applications. The ZeroMQ API is similar to the low level Socket API for communication over networks.

In contrast to a message-oriented middleware no central server is required at ZeroMQ. The sender of a message is responsible for the for routing to the right destination and the receiver of a message is responsible for queuing.

ZeroMQ has got the following notations: MQ, OMQ and ZMQ. ZeroMQ is developed by iMatix who were involved in development of AMQP. Meanwhile iMatix has withdrawn from participation and focuses entirely on ZeroMQ. The approach taken by ZeroMQ is quite elegant and enables topologies from 0 to n nodes or brokers between sender and receiver.

The abandonment of a central broker enables very low latency and high bandwidth. Thus, ZeroMQ is ideal for largest volumes of messages eg for measurement values, for real-time quotes in financial industry or for online games.

Scalability and Reliability

Basically all brokers are highly scalable, robust and reliable. Almost every project praise that as a outstanding property. Therefore, the descriptions of the broker do not separately mention these properties , unless a broker is significantly different from the others. So, this point is limited as a selection criterion.

The ActiveMQ broker has limitations in terms of scalability, robustness and reliability due to its architecture. But only at extremely high load or at thousands of queues ActiveMQ should show its limits. In practice, ActiveMQ often is more stable than its modern challenger because of its maturity.

Performance

Important in evaluating the speed of a broker is its field of application. Only searching for a fast broker usually is too simplistic. To evaluate the performance of a broker correctly you should ask yourself the following questions:

- Do I need a broker for financial transactions or online games?
- What is more important for the use: guaranteed transactional service or speed and bandwidth?
- How big may be the delay in delivery of messages?
- What size are the messages to be transmitted?
- How many queues are needed?
- How many clients simultaneously access the broker?

Properties such as persistence or transactionality cost performance and cause that messaging solutions without QoS, which keep the news to the main memory, usually are faster.

If higher throughput is required, persistence and guaranteed delivery can be disabled. In most cases the performance is despite persistence more than sufficient for many applications.

Even the somewhat dated ActiveMQ can process thousands of messages per second. Usually this is sufficient for most business applications in the fields of finance, insurance and retail. Higher demands on performance is needed in real-time processing of measured values or in online games.

If a throughput of 10 000 or more messages per second is needed, you should take one of the newer broker. The throughput of Apollo, Qpid, RabbitMQ and the other brokers lies between several hundred thousand to several

million messages per second. For example, at SpecJMS Benchmark HornetQ has delivered more than 8 million messages per second, tested on a 2 chip machine with 4 cores and 24 GB main memory. For other brokers, there are similar results.

It is very difficult to achieve an objective comparison even with standardized benchmarks because most configurations, features or protocols are different. Several brokers claimed to be the fastest. These statements usually are only valid for a short time and only for a specific scenario.

The aforementioned values allow the conclusion that performance of a broker represents a project risk just in the fewest cases. Therefore the speed is hardly a selection criterion for the benefit of a particular broker.

Practical experience:
Usually not the brokers is the bottleneck, but message consumer that are slowed down by slow backend systems or database queries.

Message Persistence

So that messages can survive the failure of a broker they have to be saved permanently on disk. For most brokers there are exchangeable Message Stores.

If messages are stored permanently, the used database greatly influences the speed of the broker. Therefore, for the storage of messages specialized databases are used. Messages have to be stored, searched by the name of a queue and be deleted. The modification of messages is not necessary. The databases optimized for these tasks, which mostly access to the hard drive directly, achieve a much better performance in persisting messages as common relational databases. The order of magnitude of the speed difference between a relational database and a special message database lies approximately at ten times.

Despite of speed costs by relational data stores many users want to cache messages in their preferred database. The JDBC store the ActiveMQ broker allows to use any relational database for which there is a JDBC driver. Of course, the Oracle database can be used for storage at ActiveMQ.

JMS Support

An important criterion when choosing a broker is the support of the *Java Message Service* standard. Many existing Java applications are using the JMS API to communicate with a Message Broker. Migration to a new broker then can occur without modifications to the application code.

In ActiveMQ and HornetQ JMS compliance is top of the list of features.

Apache Qpid comes with a JMS client API, which allows clients to connect to it over the AMQP broker. Through this API you can access other JMS brokers such as the RabbitMQ too.

Messaging Protocols

For a long time there was no standard for a messaging protocol. The Java standard JMS only describes the interface, which serves applications to communicate with a broker. The Protocol between the library and the JMS server is depending on the manufacturer.

Figure 6 shows how the same application can be operated with different message brokers using JMS. This application using the standardized JMS API through which it accesses a JMS implementation of a particular broker. In order to run the application with another broker you just have to replace the JMS client library.

Figure6: JMS Compatibility

Broker does not understand each other because of the different protocols. JMS also does not address the interoperability with other programming languages.

The *Advanced Message Queuing Protocol* is a standard that would like to fill this gap. AMQP enables collaboration of brokers and clients from different manufacturers and platforms. AMQP describes a binary

formatand the necessary application logic. Many major manufacturers set on AMQP. But there is some criticism for the way of standardization caused by the complexity of AMQP.

The approved OASIS standard version AMQP 1.0 from October 2012 is significantly different from previous versions, so that one could speak of a completely separate protocol. The previous versions 0-8, 0-9, 0-9-1 and 0-10-0 are not compatible with each other. Whether AMQP really leads to more interoperability depends on how well the AMQP version 1.0 is accepted and implemented. Support of AMQP 1.0 in ActiveMQ, Apollo and Qpid brokers and announcements for HornetQ and RabbitMQ let hope that AMQP is a success. A decisive factor may be the break with the previous too overloaded version 0-10-0. AMQP 1.0 is compared with the 0-10-0 version a little easier.

A very popular protocol for the integration of scripting languages is text-based STOMP. It's simple, performant and there are a variety of servers and clients. For STOMP there are the most client libraries, for example, C, Objective-C, PHP, Go, Python and Ruby. STOMP is that simple that you can can create a client library for the unlikely event that there is no library with just little effort. The stomp.js library for JavaScript is sizing only 7 Kbytes. To use Web pages with JavaScript as a messaging client STOMP can be combined with the WebSockets protocol. Likewise this is the way to build slim Web 2.0 applications.

The following table lists the supported protocols of each broker.

	ActiveMQ	Apollo	HornetQ	Qpid	RabbitMQ	ZeroMQ
AMQP	1.0	1.0	announced	1.0	0-8, 0-9, 0-9-1	-
MQTT	✓	✓	-	-	✓	-
OpenWire	✓	✓	-	-	-	-
REST	✓	✓	✓	-	✓	-
STOMP	✓	✓	✓	-	✓	-
STOMP over Websockets	✓	✓	✓	-	✓	-
XMPP	✓	-	-	-	Over Gateway	-

Table 1: Support for Messaging Protocols

Client Interfaces

Access to a message broker is not only reserved for Java applications. For many brokers there are client APIs in different programming languages. It is even common for client and broker to use different platforms, eg when a Java application accesses an Erlang written RabbitMQ broker.

Since AMQP has standardized the protocol on the "cable level", it is even possible to connect the AMQP client library of a broker of another broker. For this to work the AMQP versions have to match because each version of the protocol are not compatible.

Through the STOMP protocol different client platforms can be connected to a broker. STOMP is favorited to access a broker with scripting languages such as Perl, PHP and Ruby.

	ActiveMQ	Apollo	HornetQ	Qpid	RabbitMQ	ZeroQ
C	✓	-	-	✓	✓	✓
C++	-	-	-	✓	✓	✓
Erlang	-	-	-	-	✓	✓
Haskell	-	-	-	-	✓	✓
Java JMS	✓	-	✓	✓	-	-
Java proprietary	✓	-	✓	-	✓	✓
.NET	-	-	-	✓	✓	✓
Objective-C	-	-	-	-	-	✓
Perl	-	-	-	-	✓	✓
PHP	-	-	-	-	✓	✓
Python	-	-	-	✓	✓	✓
Ruby	-	-	-	✓	✓	✓

Dissemination

The ActiveMQ broker has got the largest spread by far. RabbitMQ recently has received much attention.

The embedded Google Trends chart below shows how frequently you searched for the brokers.

Behind the AMQP standard there are many heavyweights in IT and business. Among others Deutsche Börse, JPMorgan Chase, Microsoft, Progress Software, Red Hat, Software AG, U.S. Dept. of Homeland Security and VMware have been involved on the standardization. Since October 2012 AMQP is an official OASIS standard. This all leads to the conclusion that AMQP will find a large spreading.

Summary

As so often in IT the selection of a broker is no royal. With ActiveMQ, there is a proven, fast and widespread Broker whose architecture encounters its limits. The JDBC Message Store is a unique feature of ActiveMQ for data center operations.

Alternatively, there is a series of promising and cool projects, but which are still lacking somewhat in maturity and dissemination. In the projects is a lot of movement, so I assume that in 2013 many brokers will achieve sufficient stability, compatibility, and documentation.

HornetQ is useful when Enterprise Messaging with JMS support is needed. Extensive documentation describes even advanced configurations with transactions and clustering.

Who is looking for a simple broker for Web 2.0 applications, sending & receiving messages using JavaScript or who uses uses scripting languages besides Java should look at Apache Apollo.

If Erlang as a platform is not an exclusion criterion, RabbitMQ could also be interesting. Currently RabbitMQ is still missing the AMQP 1.0 compatibility.

Thomas Bayer
bayer@predic8.com

Watched Versions

The following list shows the versions which were considered for this article:

- Apache ActiveMQ 5.7.0
- Apache Apollo 1.5
- FFMQ 3.0.1
- HornetQ 2.3.0.Beta 3
- JBoss Messaging 2.0.0
- OpenJMS 0.7.7
- Qpid 0.18
- RabbitMQ 3.0.1
- ZeroMQ 3.2

List of References

AMQP Homepage
http://www.amqp.org/node/

ActiveMQ Homepage
http://activemq.apache.org/

STOMP Protocol Specification
http://stomp.github.com/stomp-specification-1.2.html

HornetQ Homepage
http://www.jboss.org/hornetq

RabbitMQ support JMS in the future?
http://rabbitmq.1065348.n5.nabble.com/RabbitMQ-support-JMS-in-the-future-td24361.html

Message Queue Evaluation Notes
http://wiki.secondlife.com/wiki/Message_Queue_Evaluation_Notes

Enabling the ActiveMQ Broker for AMQP
http://activemq.apache.org/amqp.html

Open JMS Homepage
http://openjms.sourceforge.net/

Apache Qpid Homepage
http://qpid.apache.org/

RabbitMQ Homepage
http://www.rabbitmq.com/

ZeroMQ Homepage
http://www.zeromq.org/