



TTM
Total Transaction Management

ActiveMQ 5.3 Reference Guide

PREVIEW

Total Transaction Management, LLC

An Open Source Solutions Company

570 Rancheros Drive, Suite 140

San Marcos, CA 92069

760-591-0273

www.ttmsolutions.com

NOTICE

YOU AGREE THAT THE REFERENCES, SAMPLES AND PROGRAM(S) ARE PROVIDED AS-IS, WITHOUT WARRANTY OF ANY KIND (EITHER EXPRESS OR IMPLIED). ACCORDINGLY, TOTAL TRANSACTION MANAGEMENT, LLC (TTM) MAKES NO WARRANTIES, REPRESENTATIONS OR GUARANTEES, EITHER EXPRESS OR IMPLIED, AND DISCLAIMS ALL SUCH WARRANTIES, REPRESENTATIONS OR GUARANTEES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR ANY PARTICULAR PURPOSE, AS TO:

- (A) THE FUNCTIONALITY OR NONINFRINGEMENT OF PROGRAM, ANY MODIFICATION, A COMBINED WORK OR AN AGGREGATE WORK; OR
- (B) THE RESULTS OF ANY PROJECT UNDERTAKEN USING THE PROGRAM, ANY MODIFICATION, A COMBINED WORK OR AN AGGREGATE WORK.

IN NO EVENT SHALL TTM OR ANY OF THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, CONSEQUENTIAL OR ANY OTHER DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE SAMPLES AND PROGRAMS, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. YOU HEREBY WAIVE ANY CLAIMS FOR DAMAGES OF ANY KIND AGAINST TTM OR ANY OF THE CONTRIBUTORS WHICH MAY RESULT FROM YOUR USE OF THE REFERENCES, SAMPLES, AND PROGRAM(S).

PRELIMINARY

Table of Contents

1	INTRODUCTION	1
1.1	WHAT IS ACTIVEMQ?	2
1.2	WHAT IS MOM?	2
1.3	WHY USE ACTIVEMQ?	3
1.4	WHEN AND WHERE TO USE ACTIVEMQ	4
1.4.1	Where	4
1.4.2	When	4
1.5	DOWNLOADING AND INSTALLING ACTIVEMQ	4
2	ACTIVEMQ COMPONENTS	5
3	CONNECTORS	8
3.1	CONNECTOR URIS	10
3.2	LOW-LEVEL CONNECTORS	12
3.2.1	VM	13
3.2.2	Peer	16
3.2.3	TCP	17
3.2.4	NIO	18
3.2.5	SSL	19
3.2.5.1	Setting up the SSL Key and Trust Stores	19
3.2.5.2	Starting the SSL-enabled Broker	20
3.2.5.3	Starting the SSL-enabled Client	20
3.2.6	HTTP/HTTPS	21
3.2.7	Multicast	22
3.2.8	Rendezvous	23
3.3	COMPOSITE CONNECTORS	24
3.3.1	Static	25
3.3.2	Failover	25
3.3.3	Discovery	27
3.3.4	Fanout	28
3.4	MONITORING A TRANSPORT CONNECTOR	30
4	WILDCARDS	31
5	CLIENT CONFIGURATION	31
5.1	THE JNDI.PROPERTIES FILE	31
5.2	THE LDAP-BASED JNDI	37
5.2.1	Environment Properties Hashtable	39
5.2.2	The jndi.properties File	39
5.2.3	System Properties	40
5.2.4	JNDI Operations	40
5.3	TOMCAT	42
5.4	SPRING	44
6	BROKER CONFIGURATION	44
6.1	SPECIFYING THE BROKER'S XML CONFIGURATION FILE	44
6.2	THE BROKER'S XML CONFIGURATION FILE	45
6.2.1	broker	47
6.2.2	transportConnector	48
6.2.3	networkConnector	50

6.2.4	<i>systemUsage</i>	53
6.2.4.1	<i>sendFailIfNoSpace</i>	55
6.2.5	<i>persistenceAdapter</i>	56
6.2.6	<i>destinations</i>	56
6.2.7	<i>destinationPolicy</i>	57
6.3	USEDEDICATEDTASKRUNNER	58
7	WIRE PROTOCOLS	58
8	NETWORK OF BROKERS	60
8.1	STORE AND FORWARD	60
8.2	HIGH AVAILABILITY	62
8.2.1	<i>Master/Slave Broker Configurations</i>	62
8.2.1.1	<i>Pure Master Slave</i>	62
8.2.1.2	<i>Shared File System Master Slave</i>	66
8.2.1.3	<i>JDBC Master Slave</i>	68
8.3	ISOLATING YOUR BROKER FROM OTHER BROKERS	68
9	ADVISORY MESSAGING	69
10	ACTIVEMQ MONITOR (AMON)	72
10.1	MONLET CONTAINER	72
10.2	SNMP AGENT	72
11	ADMINISTRATION	72
11.1	COMMAND LINE TOOLS	72
11.2	JCONSOLE	75
11.2.1	<i>Configuration</i>	75
11.2.2	<i>Connection</i>	76
11.2.3	<i>Operation</i>	77
11.3	WEB CONSOLE	79
11.3.1	<i>Securing Access to the Web Console</i>	83
11.4	DESTINATIONSOURCE	83
12	LOGGING	84
12.1	COMMONS-LOGGING	84
12.2	LOG4J	84
12.3	JAVA.UTIL.LOGGING	85
12.4	CONTROLLING LOGGING WITH JCONSOLE	86
12.5	CLIENT LOGGING	87
13	CONSUMER OPTIONS	87
13.1	PREFETCH LIMITS	88
13.2	SLOW CONSUMERS FOR NON-DURABLE TOPICS	90
13.3	ASYNCHRONOUS DISPATCH (DISPATCHASYNC)	94
13.4	RETROACTIVE CONSUMERS	95
13.5	EXCLUSIVE CONSUMERS	97
13.6	PRIORITIZED CONSUMERS	98
13.7	DEAD LETTER QUEUE (DLQ)	98
13.7.1	<i>individualDeadLetterStrategy</i>	99
13.7.2	<i>sharedDeadLetterStrategy</i>	100
13.7.3	<i>DiscardingDLQBroker</i>	101
13.7.4	<i>What Happens To Expired Messages?</i>	101
14	ADVANCED FEATURES	102
14.1	ASYNCHRONOUS SENDS	102
14.2	PRODUCER FLOW CONTROL (PFC)	103
14.2.1	<i>Producer Flow Control and Queues</i>	103

14.2.2	<i>Producer Flow Control and Topics</i>	104
14.2.3	<i>Synchronous Sends</i>	105
14.2.4	<i>Asynchronous Sends</i>	106
14.3	MESSAGE GROUPS.....	107
14.3.1	<i>Message Groups vs Selectors</i>	108
14.4	TOPIC MESSAGE ORDERING.....	109
14.5	BINARY LARGE OBJECTS (BLOBS).....	110
14.5.1	<i>Sending a BLOB</i>	111
14.5.2	<i>Receiving a BLOB</i>	114
14.6	ACTIVEMQ STREAMS.....	115
14.7	COMPOSITE DESTINATIONS.....	115
14.7.1	<i>Client-Side Composite Destinations</i>	115
14.7.2	<i>Broker-Side Composite Destinations</i>	117
14.7.3	<i>Selecting Composite Destinations</i>	117
14.8	VIRTUAL TOPICS.....	118
14.8.1	<i>Virtual Topics vs. Composite Destinations</i>	122
14.9	MIRRORED QUEUES.....	122
14.10	MESSAGE TRANSFORMERS.....	123
14.11	CONNECTION POOLING.....	124
14.12	SPRING SUPPORT.....	124
14.13	MESSAGE CURSORS.....	125
14.14	ENTERPRISE INTEGRATION PATTERNS.....	125
14.15	INDIVIDUAL ACKNOWLEDGE.....	125
14.16	PRIORITIZING MESSAGES.....	126
15	EXTENDING ACTIVEMQ'S FUNCTIONALITY	126
16	AUTHENTICATION AND AUTHORIZATION SERVICES	129
16.1	AUTHENTICATION.....	130
16.1.1	<i>Authentication and Authorization between Message Brokers</i>	132
16.2	AUTHORIZATION.....	133
16.2.1	<i>Controlling Access to Temporary Destinations</i>	135
16.3	CAMEL.....	135
16.4	TTM'S SECURITY PLUGINS.....	136
16.4.1	<i>File-Based Security Plugin</i>	136
16.4.2	<i>LDAP and JDBC Based Security Plugins</i>	137
17	PERFORMANCE AND SCALING	137
17.1	PERSISTENT VS. TRANSIENT MESSAGES.....	137
17.1.1	<i>Asynchronous Sends</i>	138
17.1.2	<i>Transactions</i>	138
17.2	TCP TRANSPORT.....	138
17.3	PREFETCH LIMIT.....	138
17.4	THREADS, THREAD POOLING, AND SCALING.....	140
17.4.1	<i>Blocking vs. Non-Blocking I/O</i>	140
17.4.2	<i>Thread Pooling</i>	141
17.4.2.1	<i>UseDedicatedTaskRunner</i>	141
17.4.2.2	<i>optimizedDispatch</i>	141
17.4.3	<i>Asynchronous Dispatch (dispatchAsync)</i>	141
17.4.4	<i>Asynchronous Session Dispatch (alwaysSessionAsync)</i>	141
17.4.5	<i>The KahaDB Message Store</i>	142
17.5	DUPS_OK_ACKNOWLEDGE.....	143
17.6	CONSUMER CLUSTERING.....	143
17.7	EMBEDDED BROKERS.....	143
17.8	MESSAGE COPY.....	144
17.9	OPENWIRE LOOSE ENCODING.....	144
17.10	MEMORY UTILIZATION.....	144

18	A SIMPLE JMS CHAT APPLICATION.....	145
19	STOCK WATCH LIST APPLICATION.....	146
19.1	OVERVIEW.....	146
19.2	HIGH LEVEL IMPLEMENTATION	147
19.2.1	<i>Stock Exchange Implementation.....</i>	<i>148</i>
19.2.2	<i>Stock Publisher Implementation</i>	<i>148</i>
19.2.3	<i>Message Broker Implementation</i>	<i>149</i>
19.2.4	<i>Web Server App Implementation</i>	<i>149</i>
19.2.5	<i>Web Browser Implementation.....</i>	<i>149</i>
19.3	DETAILED IMPLEMENTATION.....	151
19.3.1	<i>Server-side Spring MVC Components</i>	<i>151</i>
19.3.1.1	Data Model.....	151
19.3.1.2	View	156
19.3.1.3	Controller.....	159
19.3.1.4	Stitching it together Spring Style.....	162
19.3.1.5	Deploying the Watch-list application into Tomcat	165
19.3.1.6	Running the various Servers – Tomcat, ActiveMQ, Hsqldb.....	165
19.3.2	<i>Client Application.....</i>	<i>166</i>
19.3.2.1	Stock class	166
19.3.2.2	A Watch-list retriever – FetchShares	168
19.3.2.3	The message sending main class – TopicSendClient.....	172
19.3.2.5	Building the standalone client	178
19.3.2.6	Running the client standalone.....	178
19.3.2.7	Running the client out of Eclipse.....	178
20	JAR FILE REQUIREMENTS.....	179

1 Introduction

Apache ActiveMQ is a highly configurable, extensible, and feature-rich message-oriented middleware (MOM) system. The purpose of this document is to serve as a reference guide on how to use and configure ActiveMQ. The guide captures, what we understand to be, the more important aspects of configuring and using ActiveMQ.

This guide supplements the documentation currently available on the [Apache ActiveMQ](#) web site and there are instances where this guide refers you to specific documentation on that web site. Other sources of ActiveMQ information are the [ActiveMQ Discussion Forums](#).

The reader should have a basic understanding of MOM systems, the Java programming language, XML, and the Java Messaging Service (JMS) specification/standard.

With this guide, you're also given access to the source code for several ActiveMQ-related Java applications. The following table lists and briefly describes the applications. To download the applications click XXXX. Enter "XXXX" and "XXXXX" when prompted for the username and password, respectively. Both the username and password are case sensitive.

Application	Description
File-based Security Plugin	TTM's Security Plugin is an ActiveMQ plugin module that provides dynamically reconfigurable authentication and authorization security services.
Camel-based Framework For Message Driven POJOs	The MDP framework is a simple convenience mechanism that completely isolates a POJO from the underlying JMS provider. This Camel and Spring based framework covers topics such as dependency injection, introspection, annotations, Spring configuration files, bean auto-wiring, Camel producer/consumer templates, Camel routes, and lots more.
JMS Chat Application	A Simple JMS Chat Application that illustrates the use of the JMS and JNDI applications.
Stock Watch	The Stock Watch List Application is a composite application that represents the information flow of stock prices from a Stock Exchange through a web browser.

Thank you for purchasing a subscription for TTM's ActiveMQ Reference Guide. We hope you find the guide both informative and useful. If you have any questions, please don't hesitate to contact TTM.

TTM Sales & Support

+1 760-591-0273

sales@ttmsolutions.com

info@ttmsolutions.com

1.1 What Is ActiveMQ?

ActiveMQ is an open source MOM system that is developed and maintained by the [Apache Software Foundation](#) (ASF) community. The next section briefly describes a MOM system; to learn more about MOM systems, visit the [Middleware Resource Center](#).

ActiveMQ provides the following quality of service (QoS) features, which are expected by world-class enterprise deployments: performance, scalability, availability, reliability, transactional integrity, and security. ActiveMQ's clustering and failover technologies also provide high availability. ActiveMQ also supports a myriad of different low-level transport protocols such as TCP, SSL, HTTP, HTTPS, and XMPP.

ActiveMQ adheres to a plugin architecture that makes it an extendible messaging framework, as well as MOM system. The extendible nature of its architecture allows you to develop custom modules that are included in various parts of the core engine's processing chain. Examples of such modules are core engine plugins, transport connectors, message dispatch policies, persistence adapters, and network services.

ActiveMQ's messaging engine, or message broker, is written in the Java programming language and fully implements version 1.1 of the JMS; therefore, it presents a standards-based MOM system through which Java applications can reliably communicate messages to one another. This guide advocates the development of portable JMS applications and as such, emphasizes adherence to the JMS and the Java Naming and Directory Interface (JNDI). There are instances where the guide makes reference to ActiveMQ's native application programming interfaces (API), and in those instances the reader is warned that making direct reference to such a proprietary API will compromise the portability of their JMS client.

ActiveMQ can be deployed on any operating platform (e.g., Windows, UNIX, and Linux) that provides a compatible Java Virtual Machine (JDK 1.5 or higher).

Even though it is written in Java, with the primary goal of implementing the JMS API, ActiveMQ also supports other programming languages (e.g., C#, Ruby, Python, C/C++). This document focuses on JMS clients and does not cover the non-Java clients. Please refer to the ActiveMQ web site for more information on ActiveMQ's support for non-Java programming languages.

You should visit the [FAQ](#) on the ActiveMQ web site as it covers a broad range of ActiveMQ topics.

1.2 What is MOM?

MOM is a specific type of messaging middleware that facilitates communications between loosely coupled distributed applications. MOM is more closely identified with providing *asynchronous* communications, via queues, between the loosely coupled applications. So within the MOM framework, messages are sent to and delivered from a message queue. MOM clients send a message to a queue and the message remains in the queue until another MOM client retrieves the message from that queue. One advantage to this asynchronous messaging model is

that the client retrieving the message does not have to be available when the message is sent to the queue and can instead retrieve the message at any time. This is sometimes referred to as deferred communications.

All of the above is opposed to a more tightly-coupled synchronous communications model where the sender and receiver of a message must be available at the same time in order to successfully communicate with one another. One example of a tightly coupled distributed system requiring synchronous communications is Remote Method Invocation (RMI). With RMI, the sender requires the receiver to be available when it sends the message; if not, the corresponding remote method invocation fails.

Even though MOM is more closely identified with asynchronous communications, most MOM implementations, including ActiveMQ, can also accommodate the more tightly-coupled synchronous communications paradigm. This is typically performed via the [Request-Reply](#) messaging pattern.

1.3 Why Use ActiveMQ?

ActiveMQ is the most popular open source software (OSS) MOM today, and is rapidly becoming the de-facto MOM for OSS-based Service Oriented Architecture (SOA) deployments.

The following are advantages of employing OSS and ActiveMQ:

- 1) OSS advantages:
 - a. OSS is more cost effective than proprietary products; there are no royalty license fees.
 - b. OSS is now accepted due to the success of Linux and the numerous projects (including ActiveMQ) that comprise the Apache Software Foundation (ASF).
 - c. OSS provides customers freedom from being “locked in” to proprietary providers of expensive software products, licenses, and support.
 - d. OSS provides more flexibility for support and current status through third party professional services providers and public user forums.
 - 2) AMQ advantages and features:
 - a. ActiveMQ is OSS
 - b. ActiveMQ can interoperate (via JMS bridging) with other JMS providers (e.g., IBM’s WebSphere MQ (formerly MQSeries), Progress’ SonicMQ, and Oracle’s WebLogic), and supports clients written in C/C++, C#, Ruby, Perl, and PHP. This provides a path for enterprises to embrace open source ESB for future implementations, regardless of past technology commitments.
 - c. ActiveMQ offers proven scalability, availability, and performance that will grow with the customer’s requirements.
 - d. ActiveMQ is standards based; it supports the JMS 1.1 open standard.
 - e. The ActiveMQ message broker is written in the Java language, and is thus very portable.
 - f. In combination with Apache Camel, ActiveMQ can facilitate a wide range of messaging patterns.
 - g. ActiveMQ is continually supported and updated by the ASF and the user community.
-

- h. ActiveMQ is a fast and feature-rich open source JMS message broker primarily targeted for loosely coupled, distributed application environments. It provides message persistence, guaranteed message delivery, and can be highly scalable through clustering, peer-to-peer and federated network.
- i. ActiveMQ can use JDBC for persistence, and when combined with journaling, can provide high performance persistence.
- j. ActiveMQ supports a variety of transport protocols such as TCP, HTTP, XMPP, SSL, UDP and multicast.
- k. ActiveMQ is Spring-based and configured using dependency injection.

1.4 When and Where to Use ActiveMQ

1.4.1 Where

ActiveMQ is becoming the preferred OSS MOM for any modern enterprise application architecture, and can be used to implement a SOA framework co-existing and supporting historical client-server, publish and subscribe, and XML-oriented applications.

ActiveMQ can be integrated with a number of platforms/frameworks including: Geronimo, Spring, and Apache Tomcat.

1.4.2 When

ActiveMQ provides a clean application agnostic interface when you need applications to communicate by writing and retrieving application-specific data (messages) to/from queues, without having a private, dedicated, logical connection to link them.

ActiveMQ is the preferred Open Source MOM when you need:

- Availability: Transparent load balancing, failover, and recovery
- Interoperability: With many various message stores including JDBC
- Manageability: Can be administered with JMX
- Performance: Approaching or competing with proprietary solutions
- Reliability: Guaranteed once-and-only-once message delivery
- Scalability: Clustering and load balancing features
- Security: SSL, HTTPS, and authentication and authorization.

1.5 Downloading and Installing ActiveMQ

To download and install ActiveMQ on your operating platform, follow the instructions described in the “ActiveMQ Getting Started Guide”, which can be found at <http://activemq.apache.org/getting-started.html>

If you have downloaded multiple distributions of ActiveMQ, set the `ACTIVEMQ_HOME` environment variable to point to the distribution you will be using. For example,

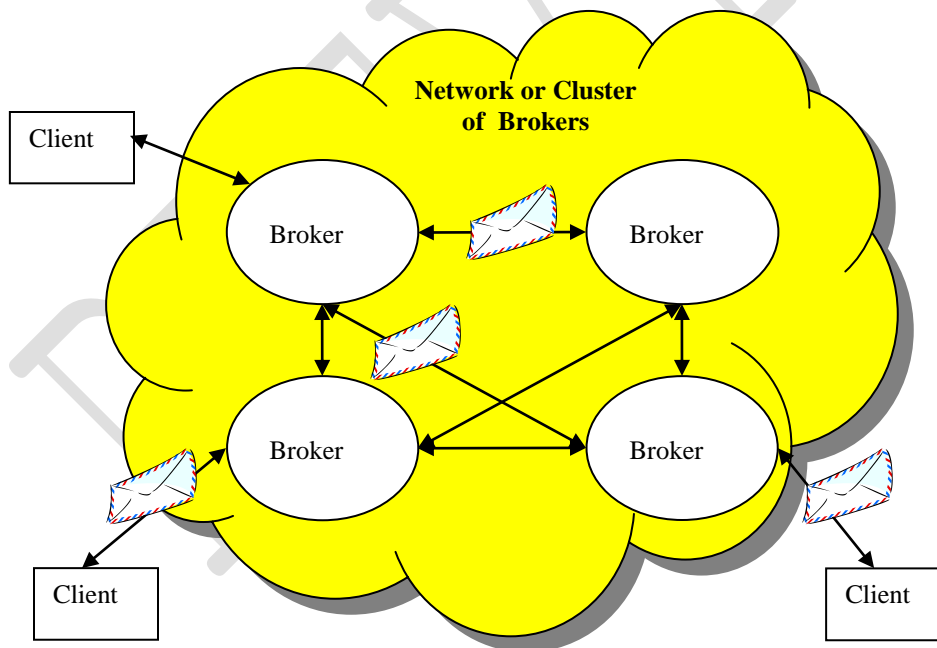
```
> export ACTIVEMQ_HOME=$HOME/apache-activemq-5.0
```

Ensure that, at a minimum, the following jar file is in your client application's CLASSPATH: `activemq-all-<version>.jar`. See section 20 for a list and description of other jar files that your particular deployment may require.

2 ActiveMQ Components

Some of the more important components of the ActiveMQ messaging framework are the client (application), message, destination, and message broker.

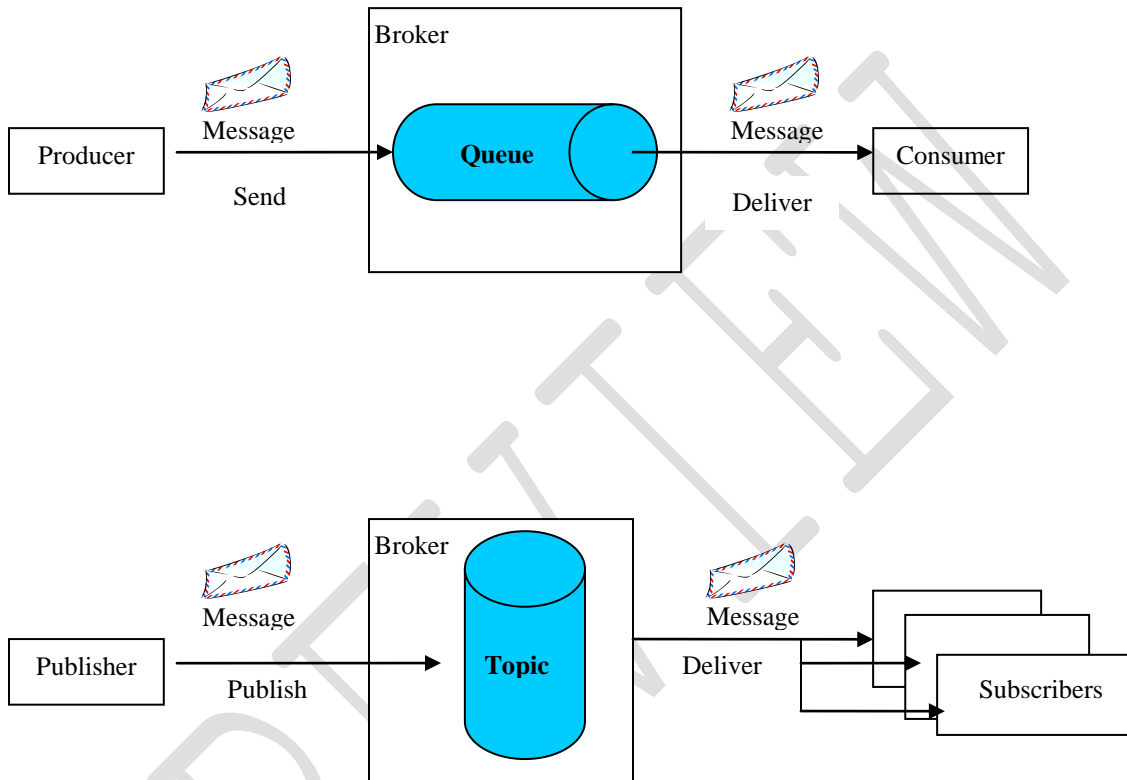
The client, which is an application component that uses the services provided by the message broker, can be further categorized as either a message *producer* and/or *consumer*. A producer creates a message, which it then gives to the message broker for routing and delivery to a particular destination. Consumers retrieve messages from the destinations to which they have been routed. A destination can therefore be viewed as a logical channel through which clients communicate with one another. It is the responsibility of the ActiveMQ message broker or network of brokers (NoB) to not only route the message to the correct destination, but to also ensure adequate quality of services such as reliability, persistence, security, and high availability. An ActiveMQ NoB can take on different network topologies such as hub-n-spoke, ring, peer-to-peer, etc.



Many times an analogy is drawn between a MOM system and a postal service. That is, the client is a postal customer, the postal system is a broker or NoB, the message is a letter, and the destination is a post office box.

A destination, which can be either a *queue* or a *topic*, is maintained by the message broker (a broker can maintain many destinations). Queues are used by producers to send a message to a

consumer (1:1 relationship), while topics are used by a producer to send a message to one or more consumers (1:N relationship). A queue is often-times referred to as a point-to-point messaging channel, while a topic is referred to as a publish-subscribe messaging channel. Producers that send messages to topics are more commonly referred to as *publishers* and consumers that retrieve messages from topics are referred to as *subscribers*. To recap, a message is read from a queue by only one consumer, while one or more consumers (subscribers) can read a message from a topic.

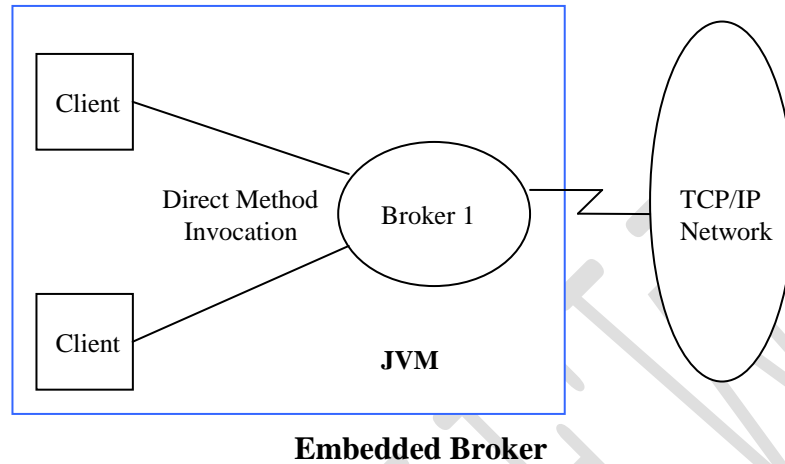


As previously mentioned, the message broker is the ActiveMQ component that accepts messages from producers and delivers those messages to their corresponding target destination (queue or topic). The broker or NoB is also responsible for delivering or dispatching messages from the destination to one or more consumers. In ActiveMQ vernacular, the term “dispatching” is more commonly used to describe the delivery of messages from the destination to the consumer. In the process of routing and dispatching messages, the broker provides quality of service features such as guaranteed delivery, high availability, security, and reliability.

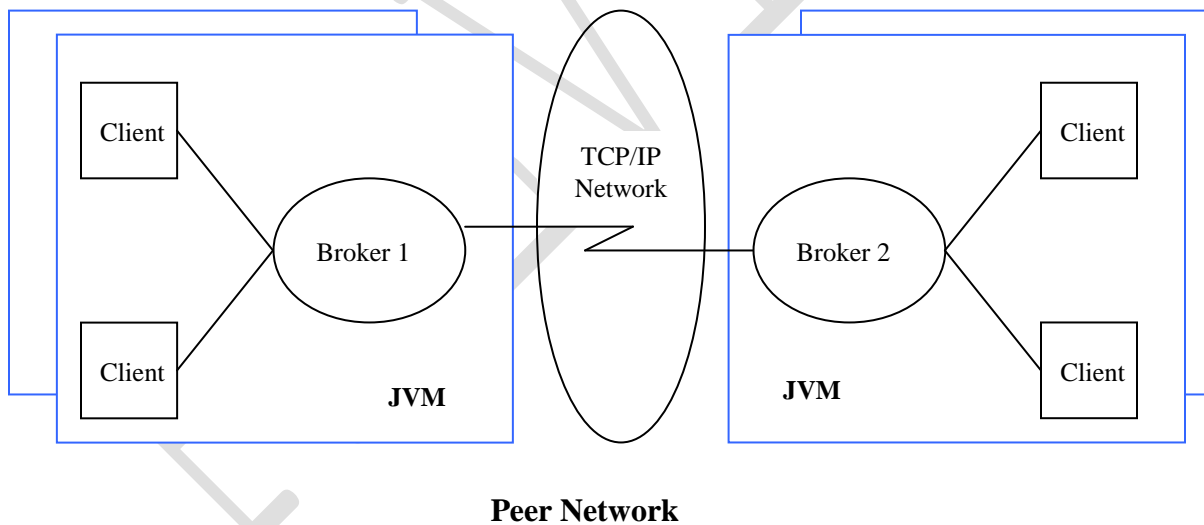
There are two basic types of ActiveMQ brokers: embedded and standalone.

An embedded broker executes within the same JVM process as the Java clients that are using its services. There may be one or more clients residing within a single JVM, each executing within its own thread(s) of execution; all clients access the same embedded broker. The clients communicate with their embedded broker via direct method invocation (DMI) as opposed to serializing command objects (messages) across a TCP/IP-based transport connector. One advantage of using an embedded broker is that if the network fails, its embedded clients can still use the services of the broker. For example, a producing client can still send messages to the

broker and any messages that need to be forwarded on to another broker will be held and/or persisted by the broker until the network is once-again made available. Another advantage is increased performance, because the broker's embedded clients communicate with the broker via DMI instead of across a TCP/IP connection.

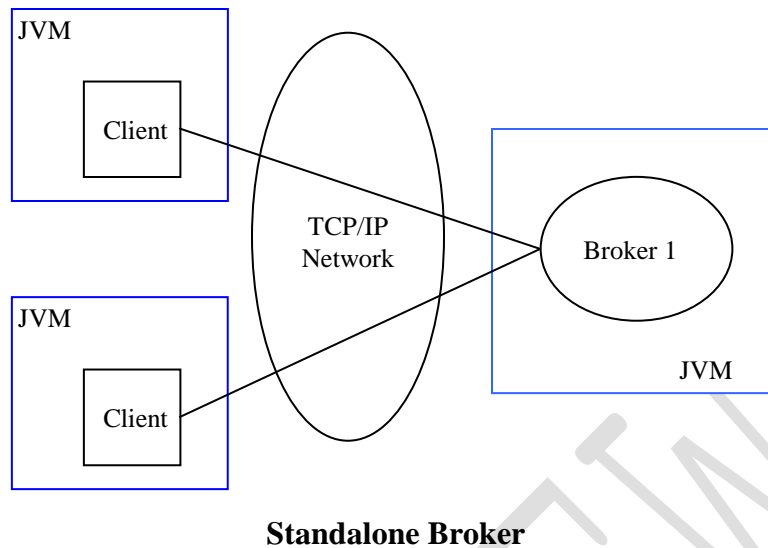


Embedded brokers can connect to other embedded brokers to form what is referred to as a “peer” network. Peer networks provide better performance because there is only one network hop involved when sending a message from a producer/publisher to a consumer/subscriber.



A client starts an embedded broker via the vm or peer transport connectors (see sections 3.2.1 and 3.2.2).

Embedded brokers can also listen for and initiate connections to standalone or non-embedded brokers. Unlike an embedded broker, a *standalone* broker is one that does not have its clients co-residing in its JVM and communicates with its clients through network-based transport connectors, which are covered in the next section.



3 Connectors

Within the ActiveMQ nomenclature, the terms ‘*transport connector*’ and ‘*network connector*’ are significant. These connectors represent network communication channels through which the clients communicate with their respective brokers and brokers communicate with one another. The underlying wire format protocols used through these communication channels are called, “OpenWire”, “Stomp”, “REST”, and “XMPP” (see section 7); the default protocol being OpenWire. So when a client invokes a JMS operation (e.g., message send), the ActiveMQ client libraries will wrap that operation into an OpenWire command object and then send or serialize the command object, via the underlying ‘transport connector’, to the broker.

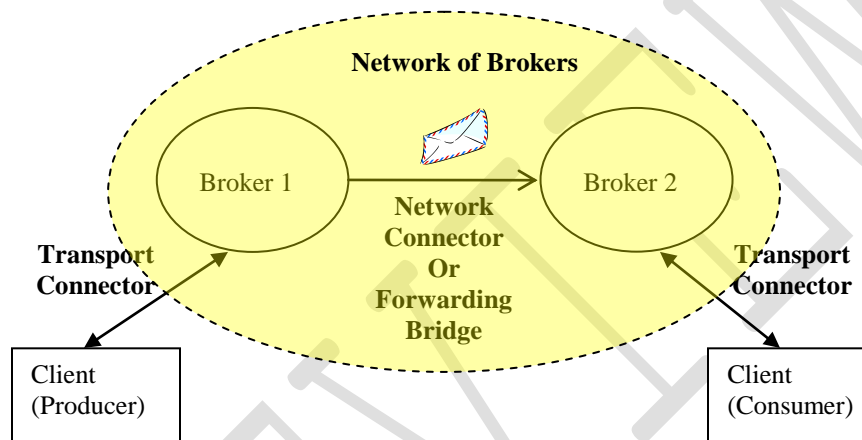
The following table lists the ActiveMQ supported wire protocols along with links to additional information for the protocols.

Wire Protocol	Additional Information
OpenWire	http://activemq.apache.org/openwire.html http://activemq.apache.org/openwire-version-2-specification.html
Stomp	http://activemq.apache.org/stomp.html
REST	http://activemq.apache.org/rest.html http://rest.blueoxen.net/cgi-bin/wiki.pl
XMPP	http://activemq.apache.org/xmpp.html http://xmpp.org/

A ‘transport connector’ is used by a client to establish a bidirectional communication channel with a broker. It is also used by a broker to listen for and accept network connection requests from clients and other brokers.

A ‘network connector’ is used by a broker to establish a communications channel to another broker. This type of channel is also referred to as a “*forwarding bridge*”.

Transport and network connectors are typically specified through the client and broker’s external configuration files.



When a broker (say broker 1) establishes or initiates a network connection with another broker (say broker 2), the resulting connection serves as a unidirectional forwarding bridge that is used by broker 1 to forward messages on to broker 2. In this case, broker 1 is referred to as the producing broker, whilst broker 2 is referred to as the consuming broker. For example, if a broker has producers, but no consumers, it may use one or more forwarding bridges to forward messages on to those brokers that have appropriate consumers. If a broker has multiple forwarding bridges, with appropriate consumers at the other ends of the bridges, it will load balance messages across the bridges.

As described above, the *default* behavior of a network connector is to construct a unidirectional forwarding bridge between two message brokers. So if broker 1 initiates a network connection with broker 2, then the resulting connection can only accommodate messages flowing from broker 1 to broker 2. The connection cannot accommodate messages flowing from broker 2 to broker 1. In effect, broker 1 serves as the message producer and broker 2 the message consumer. This default behavior could not, for example, accommodate two clients implementing a request-reply messaging pattern, because the reply from the consumer could not flow back to the producer. However, starting with version 5.0 of ActiveMQ, you can override this default behavior so that the network connection can accommodate messages flowing in either direction. In other words, brokers at either end of the channel can both produce and consume messages to and from one another. This type of *bidirectional* network connection is referred to as a ‘duplex’