ActiveMQ 5.3
Reference Guide

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

Table of Contents

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,

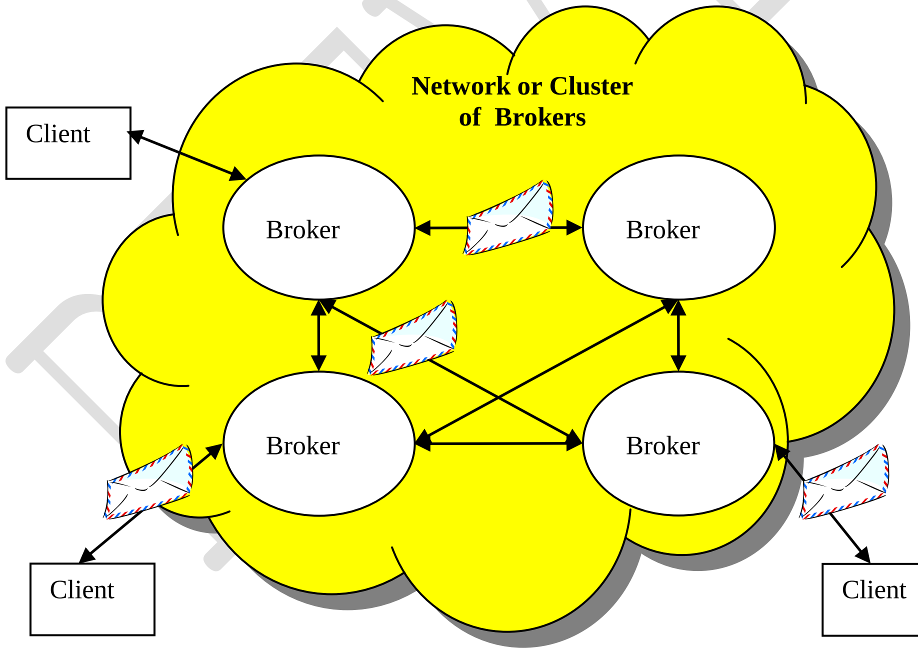
```
> export ACTIVEMQ_HOME=$HOME/apache-activemq-5.0
```

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

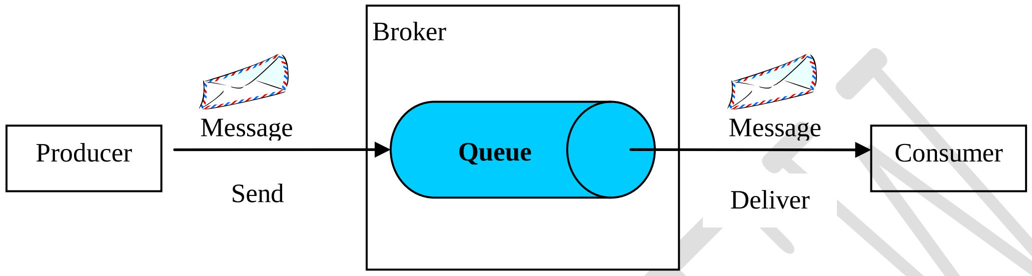
Some of the more important components of the ActiveMQ messaging framework are the client (application), message, destination, and message broker.

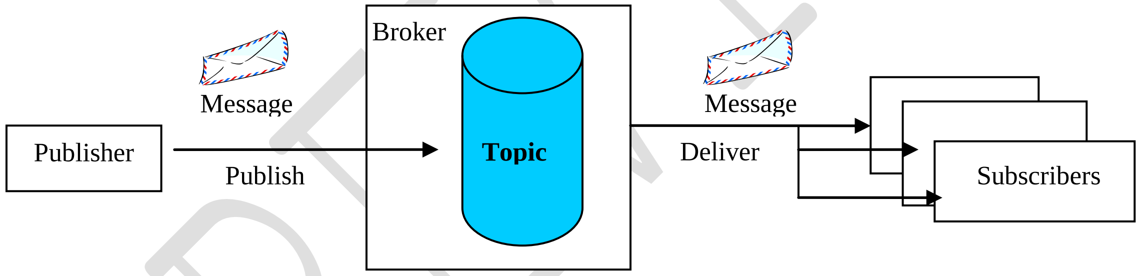
The client, which is an application component that uses the services provided by the message broker, can be further categorized as either a message *producer* and/or *consumer*. A producer creates a message, which it then gives to the message broker for routing and delivery to a particular destination. 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.

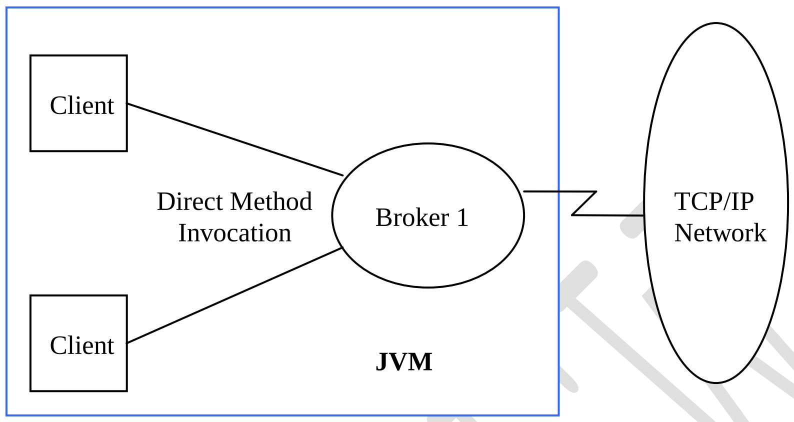




As previously mentioned, the message broker is the ActiveMQ component that accepts messages from producers and delivers those messages to their corresponding target destination (queue or topic). The broker or NoB is also responsible for delivering or dispatching messages from the destination to one or 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.

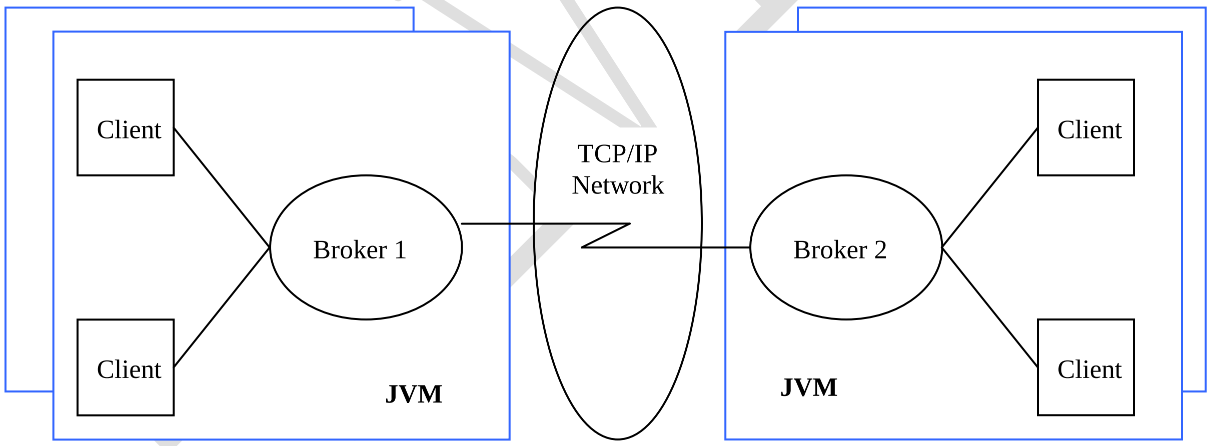
There are two basic types of ActiveMQ brokers: embedded and standalone.

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 Broker

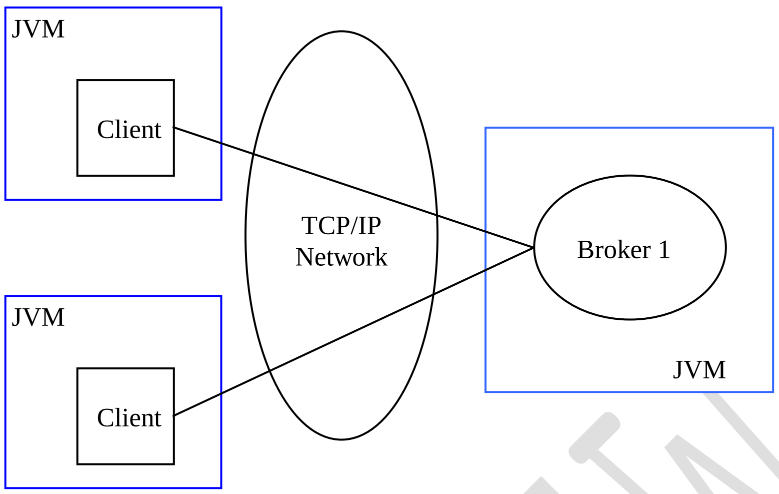
Embedded brokers can connect to other embedded brokers to form what is referred to as a "peer" network. Peer networks provide better performance because there is only one network hop involved when sending a message from a producer/publisher to a consumer/subscriber.



Peer Network

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.



Standalone Broker

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.

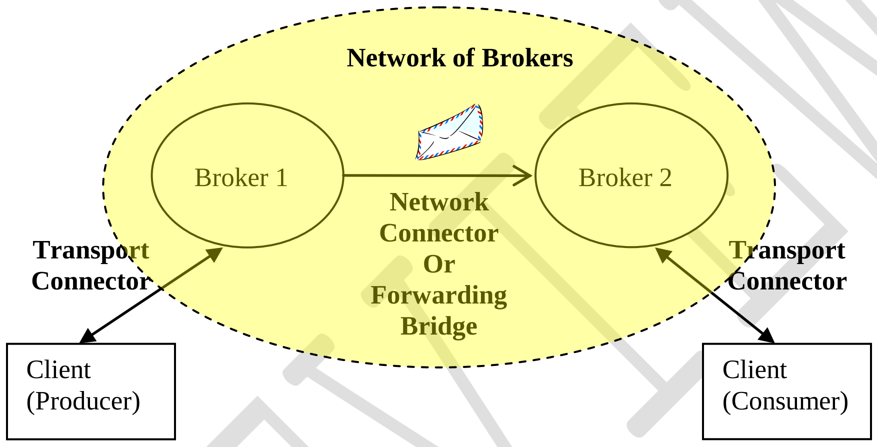
The following table lists the ActiveMQ supported wire protocols along with links to additional information for the protocols.

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’