ActiveMQ in Action

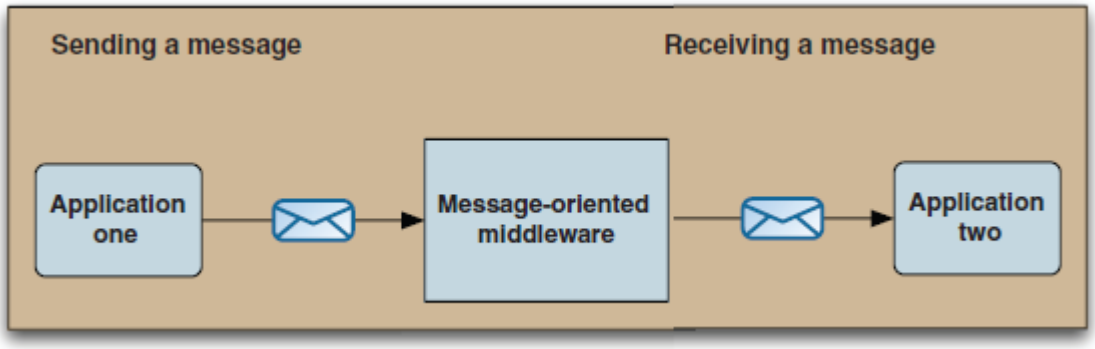
1. Introduction to Apache ActiveMQ

MOM: Message-Oriented Middleware

ActiveMQ provides the benefits of loose coupling for application architecture. Applications sending messages to ActiveMQ aren't concerned with how or when the message is delivered. Consuming applications have no concern with where the messages originated or how they were sent to ActiveMQ.

ActiveMQ acts as a middleman, allowing heterogeneous integration and interaction in an asynchronous manner.

Coupling refers to the interdependence of two or more applications or systems. Using RPC, when one application calls another one, the caller is blocked until the callee returns control to the caller.



When to use ActiveMQ

- *Heterogeneous application integration* (cross-language)
- *As a replacement for RPC*
- *To loosen the coupling between applications*
- *As the backbone of an event-driven architecture*: when an order is place at Amazon, it is accepted and acknowledged immediately. The rest of the steps in the process are handled asynchronously. If an error occurs, the user is notified via email. This allows massive scalability and high availability.
- *To improve application scalability*

2. Understanding message-oriented middleware and JMS

ActiveMQ is a MOM product that provides asynchronous messaging for business systems. The purpose of enterprise messaging was to transfer data among disparate systems by sending messages from one system to another.

Examples of products: *WebSphere MQ, Sonic MQ, TIBCO, Apache Active MQ*

The overall idea behind a MOM is that it acts as a message mediator between message senders and receivers.

JMS provides an API for enterprise messaging. **`MessageProducer`** class sends messages to a destination. **`MessageConsumer`** class consumes messages from a destination. The **`MessageListener`** `onMessage()` is invoked as messages arrive on the destination. The **JMS provider** is the vendor-specific MOM that implements the JMS API. The JMS **message** transmits the data and the events.

Delivery Mode: Persistent (message delivered once and only once) <> Non-Persistent (at most once).

Properties: custom, JMS defined, provider-specific.

Selectors filter the message, using Boolean logic - only for the header and the properties, not the payload.

```
String selector = "SYMBOL = 'AAPL' AND PRICE > " + getPrice();
MessageConsumer consumer = session.createConsumer(destination, selector);
```

Message Types: `Message`, `TextMessage`, `MapMessage`, `BytesMessage`, `StreamMessage`, `ObjectMessage`.

Domains: point-to-point [queue], publish/subscribe [topic]

Administered objects: `ConnectionFactory` and `Destination`

Workflow:

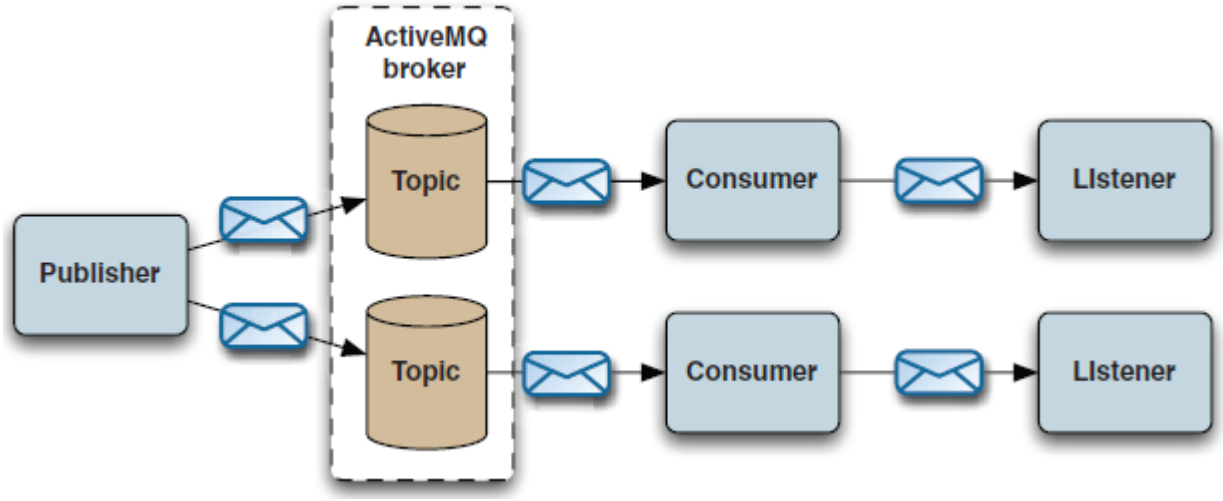
```
1 Acquire a JMS connection factory
2 Create a JMS connection using the connection factory
3 Start the JMS connection
4 Create a JMS session from the connection
5 Acquire a JMS destination
6 Create a JMS producer
      a Create a JMS producer
      b Create a JMS message and address it to a destination
7 Create a JMS consumer
      a Create a JMS consumer
      b Optionally register a JMS message listener
8 Send or receive JMS message(s)
9 Close all JMS resources (connection, session, producer, consumer...)
```

To receive a message synchronously: `consumer.receive(1000);`

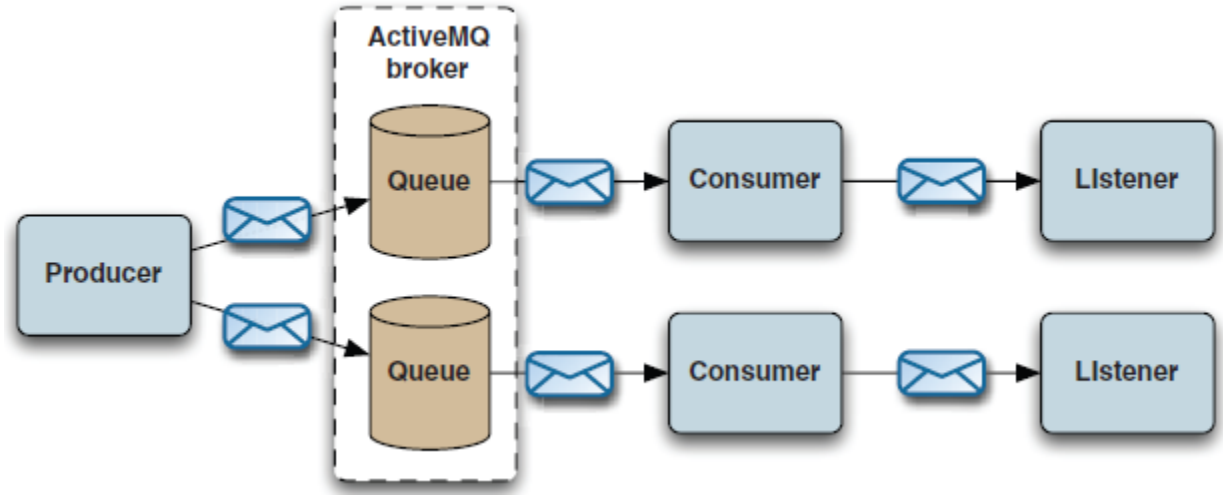
Asynchronously: `consumer.setMessageListener(MessageListener);`

3. The 'ActiveMQ in Action' examples

Example 1: Stock Portfolio



Example 2: Job Queue



4. Connecting to ActiveMQ

Connector: mechanism that provides client-to-broker communications (transport connector) or broker-to-broker communications (network connector).

Connector URI

URI = compact string of characters for identifying an abstract or a physical resource

`tcp://localhost:61616` = create a TCP connection to the localhost on port 61616.

Transport Connectors

Mechanism used to accept and listen to connection from clients.

```
<transportConnectors>
      <transportConnector name="openwire" uri="tcp://localhost:61616"
discoveryUri="multicast://default"/>
      <transportConnector name="ssl" uri="ssl://localhost:61617"/>
      <transportConnector name="stomp" uri="stomp://localhost:61613"/>
      <transportConnector name="xmpp" uri="xmpp://localhost:61222"/>
</transportConnectors>
```

Summary

Protocol	Description
TCP	Transmission Control Protocol. Default network protocol for most use cases. Highly reliable host-to-host protocol. **`tcp://hostname:port`**`?key=value&key=value`
NIO	New I/O protocol. Provides better scalability for connections from producers to the broker. **`nio://hostname:port`**`?key=value`
UDP	User Datagram Protocol. To deal with the firewall between clients and the broker. UDP does not guarantee packet ordering and uniqueness, and is connectionless. **`udp://hostname:port`**`?key=value`

SSL	Secure Socket Layer Protocol. Allow secure communication.
	`ssl://hostname:port``?key=value`
	System parameters:
	`-Djavax.net.ssl.keyStore=client.ks`
	`-Djavax.net.ssl.keyStorePassword="password"`
	`-Djavax.net.ssl.trustStore=client.ts`

HTTP(S)	To deal with the firewall between clients and the broker **`http[s]://hostname:port`**`?key=value`

VM	For communication within the same Java Virtual Machine (JVM) **`vm://brokerName`**`?key=value`

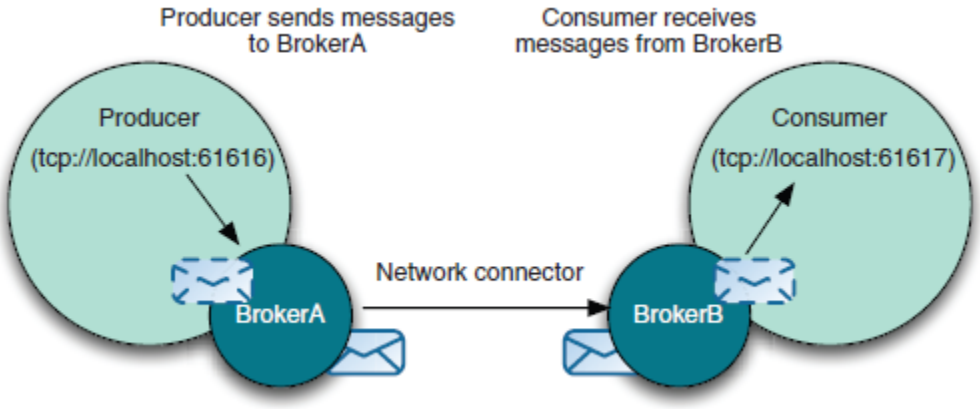
`Keystores` = hold your private certificates with their corresponding private <> `Truststores` = hold other applications trusted certificates.

A **network of brokers** creates a cluster composed of multiple ActiveMQ instances that are interconnected to meet more advanced scenarios.

Discovery is a process of detecting remote broker services.

A *static network connector* is used to create a static configuration of multiple brokers:

`static:(uri1,uri2,uri3,...)``?key=value`



IP multicast is a network technique for transmission to a group of interested receivers. The group address is an IP address in the range of 224.0.0.0 to 239.255.255.255. Brokers use the multicast protocol to advertise their services and locate the services of other brokers. Clients use multicast to locate brokers and establish a connection with them.

`multicast://ipadaddress:``port?key=value`

The *discovery* protocol allows client to discover brokers and randomly choose one to connect to.

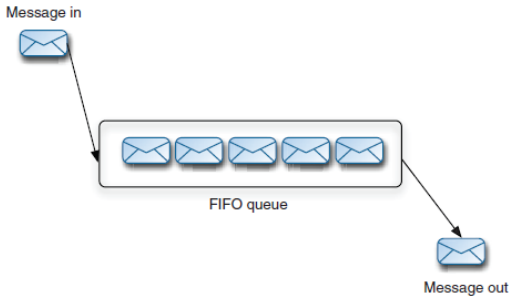
A *peer protocol* allows the creation of a network of embedded brokers.

A *fanout protocol* is used by clients to connect to multiple brokers and replicate operations between them.

5. ActiveMQ message storage

Delivery modes: persistent and non-persistent. Persistent message must be logged to a stable storage.

Storage for queues: FIFO



Store for topics:

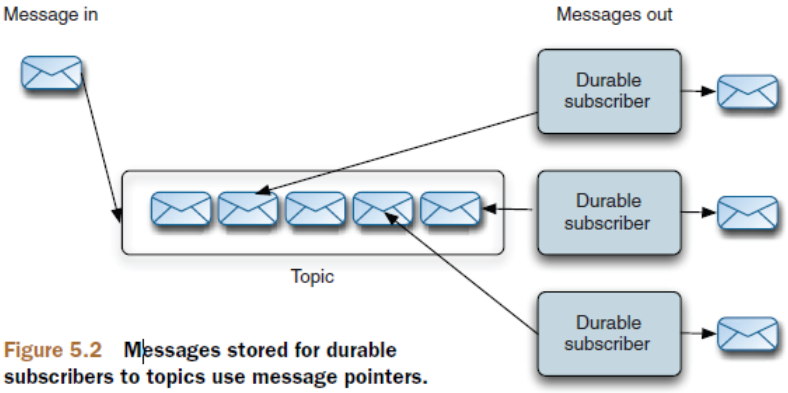


Figure 5.2 Messages stored for durable subscribers to topics use message pointers.

Recommended message store for ActiveMQ: **`KahaDB`**

The **AMQ** message store, like `KahaDB`, is a combination of a transactional journal for reliable persistence (to survive system crashes) and high-performance indexes, which makes this store the best option when message throughput is the main requirement for an application.

The **JDBC** message store allows messages to be store into databases. Default database: Apache Derby.

The **memory** message store holds all persistent messages in memory.

Recovery policies allow for fine-tuning the duration and type of messages that are cached for non-durable topic consumers.

6. Securing ActiveMQ

The easiest way to secure the broker is through the use of authentication credentials placed directly in the broker's XML configuration file.

```
<plugins>
<simpleAuthenticationPlugin>
      <users>
            <authenticationUser username="admin" password="password"
                  groups="admins,publishers,consumers"/>
            <authenticationUser username="publisher" password="password"
                  groups="publishers,consumers"/>
            <authenticationUser username="consumer" password="password"
                  groups="consumers"/>
            <authenticationUser username="guest" password="password"
                  groups="guests"/>
      </users>
</simpleAuthenticationPlugin>
</plugins>
```

To supply the user name / password:

```
factory = new ActiveMQConnectionFactory(brokerURL);
connection = factory.createConnection(username, password);
```

JAAS plug-in

The JAAS plug-in provides the same functionalities than the simple authentication plug-in, but uses standardized Java mechanism.

```
<plugins>
      <jaasAuthenticationPlugin configuration="activemq-domain" />
</plugins>
```

Authorization

ActiveMQ provides two levels of authorization: destination/operation-level and message-level.

There are 3 types of user-level operations with JMS destinations: read/write/admin.

```
<authorizationEntry topic="STOCKS.>" read="consumers" write="publishers"
admin="publishers" />
```

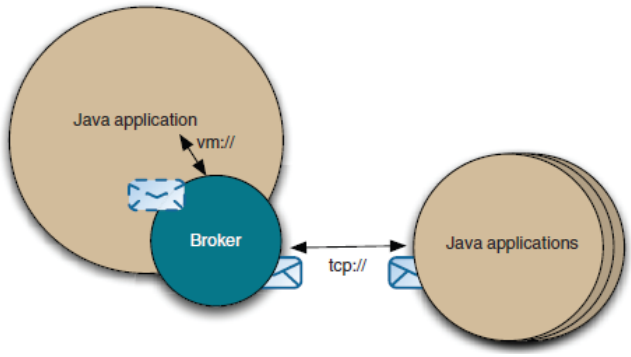
Message level authorization controls access to particular messages in a destination. Use the `MessageAuthorizationPolicy` with the method `isAllowedToConsume`.

SSL Certificates

Certificates can be used to avoid storing client credentials using plain user names and passwords.

7. Creating Java applications with ActiveMQ

A fully configured broker can serve clients from the same application (using the VM protocol) as well as clients from remote applications.



Embedding ActiveMQ using Java

Starting point: `org.apache.activemq.broker.BrokerService`
`BrokerFactory`: utility to create a broker instance using an ActiveMQ URI.

The `xbean` URI scheme tells the broker to search for the given XML configuration file in the classpath.

Embedding ActiveMQ with Spring

To use a pure Spring XML syntax with ActiveMQ, define the `BrokerService` as a bean in the Spring configuration file.

By default, ActiveMQ uses Spring and Apache `XBean` for its internal configuration purposes. `XBean` provides the ability to define and use a custom XML syntax.

```
org.apache.xbean.spring.context.FileSystemXmlApplicationContext context = new
FileSystemXmlApplicationContext(config);
```

Implementing request / reply with JMS

For a high level, a request / reply scenario involves an application that sends a message (request) and expects to receive a message (reply) in return. Some of the most scalable systems in the world are implemented using asynchronous processing.

In the diagram below, the producer creates a request in the form of a JMS message and sets a couple of important properties: the correlation ID and the reply destination. The client configures a consumer to listen on the reply destination.

Second, a worker receives the request, processes it, and sends a reply message using the destination named in the `JMSReplyTo` property of the request message.

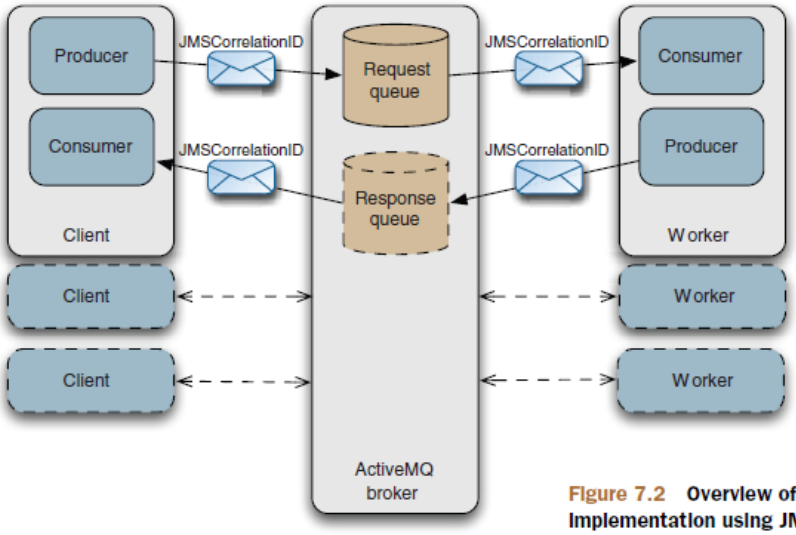


Figure 7.2 **Overview of a request/reply implementation using JMS**

To increase the scalability of the process, just add additional workers to handle the load.

To assign correlation ID:

```
response.setJMSCorrelationID(message.getJMSCorrelationID());
```

On the client side, send request:

```
TextMessage txtMessage = session.createTextMessage();
txtMessage.setText(request);
txtMessage.setJMSReplyTo(tempDest);
txtMessage.setJMSCorrelationID(correlationId);
this.producer.send(txtMessage);
```

Wait for Reply:

```
tempDest = session.createTemporaryQueue();
consumer = session.createConsumer(tempDest);
consumer.setMessageListener(this);

public void onMessage(Message message) {
      …
}
```

Writing JMS client with Spring

Configuring JMS connections

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
      <property name="userName" value="admin" />
      <property name="password" value="password" />
</bean>
```

Configuring JMS destinations

```
<bean id="cscoDest" class="org.apache.activemq.command.ActiveMQTopic">
      <constructor-arg value="STOCKS.CSCO" />
</bean>
```

Creating JMS consumers

```
<bean id="cscoConsumer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
      <property name="connectionFactory" ref="jmsConnectionFactory" />
      <property name="destination" ref="cscoDest" />
      <property name="messageListener" ref="portfolioListener" />
</bean>
```

Creating JMS producers

The `JmsTemplate` class is a convenience class for sending messages.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
<property name="connectionFactory" ref="pooledJmsConnectionFactory" />
</bean>

<bean id="stockPublisher" class="org….book.ch7.spring.SpringPublisher">
      <property name="template" ref="jmsTemplate" />
      <property name="destinations">
      <list>
            <ref local="cscoDest" />
            <ref local="orclDest" />
      </list>
      </property>
</bean>

Destination destination = destinations[idx];
template.send(destination, getStockMessageCreator(destination));
```

8. Integrating ActiveMQ with Application Servers

Application servers provide a container architecture that accepts the deployment of an application and provides an environment in which it can run. First type of AS implements the Java Servlet Specifications = ***Web container*** (ex: Tomcat, Jetty). Second type implements the Java EE Specifications = ***Java EE container*** (ex: Geronimo, JBoss, WebLogic, WebSphere).

Spring can be used to start ActiveMQ and to provide access to the JMS destinations. It is also possible to create an instance of the broker with a JMS connection.

Java AS provide a JNDI implementation that expose objects to be used by applications deployed to the container. *Local JNDI* is used to configure objects that will be exposed to a specific application, while *global JNDI* is for any application in the entire web container.

web.xml:

```
<resource-ref>
      <res-ref-name>jms/ConnectionFactory</res-ref-name>
      <res-type>org.apache.activemq.ActiveMQConnectionFactory</res-type>
      <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
      <res-ref-name>jms/FooQueue</res-ref-name>
      <res-type>javax.jms.Queue</res-type>
      <res-auth>Container</res-auth>
</resource-ref>
```

The `<resource-ref>` elements reference the JNDI resources that are registered with the AS. This make the resources available to the web application.

Spring application context:

```
<jee:jndi-lookup id="connectionFactory"
jndi-name="java:comp/env/jms/ConnectionFactory"
cache="true"
resource-ref="true"
lookup-on-startup="true"
expected-type="org.apache.activemq.ActiveMQConnectionFactory"
proxy-interface="javax.jms.ConnectionFactory" />
```

The `<jndi-lookup>` elements utilize Spring to perform a JNDI lookup of the noted resource. The resources are injected into the `messageSenderService` bean.

```
<bean id="messageSenderService"
      class="org....book.ch8.jms.service.JmsMessageSenderService"
      p:jmsTemplate-ref="jmsTemplate" />
```

This bean is used by the web application to send a JMS message.

Integrating with Tomcat

The JNDI resources are defined in a file named `META-INF/context.xml`.

```
<Context reloadable="true">
<Resource auth="Container"
      name="jms/ConnectionFactory"
      type="org.apache.activemq.ActiveMQConnectionFactory"
      description="JMS Connection Factory"
      factory="org.apache.activemq.jndi.JNDIReferenceFactory"
      brokerURL="vm://localhost?brokerConfig=xbean:activemq.xml"
      brokerName="MyActiveMQBroker"/>
<Resource auth="Container"
      name="jms/FooQueue"
      type="org.apache.activemq.command.ActiveMQQueue"
      description="JMS queue"
      factory="org.apache.activemq.jndi.JNDIReferenceFactory"
      physicalName="FOO.QUEUE"/>
</Context>
```

If a resource has been defined in a <`Context`> element it is not necessary for that resource to be defined in /WEB-INF/`web.xml`. However, it is recommended to keep the entry in /WEB-INF/`web.xml` to document the resource requirements for the web application.

9. ActiveMQ messaging for other languages

STOMP: Streaming Text Oriented Messaging Protocol

Open Wire protocol: efficient binary protocol in bandwidth and performance

10. Deploying ActiveMQ in the enterprise

High availability: run multiple ActiveMQ brokers on different machines, so that if one fails, a secondary one can take over. Master/slave: one broker takes the role of the master and more wait for the master to fail.

A failure of the master is detected by loss of connectivity from the slave to the master.

Shared nothing master/slave: each broker has its own unique message storage. All messages are replicated from the master to the slave. A master is allowed only one slave, and a slave itself can't have another slave. Solution acceptable when some down time on failure is acceptable. Manual intervention by an administrator will be necessary to configure a new slave for the new master.

The client will use the fail-over transport:
`failover://(tcp://masterhost:61616,tcp://slavehost:61616)?randomize=false`

Shared storage master/slave: multiple brokers can connect to the shared message store (a DB or a share file system) with only one broker active at a time. No manual intervention is required to maintain the integrity of the application. No limitation on the number of slave brokers. The new master will be the slave able to grab the lock on the DB.

Store and forward: messages are always stored in the local broker, and forwarded across the network to another broker.

Vertical Scaling: technique used to increase the number of connections that a single broker can handle. By default, ActiveMQ uses blocking I/O for connections (= one thread per connection). Non-blocking I/O reduces the number of threads.

Horizontal Scaling: technique to increase the number of ActiveMQ brokers available for the applications. Introduce some latency because messages may have to pass through multiple brokers before being delivered to a consumer.

Client-side traffic partitioning is a hybrid of vertical and horizontal partitioning.

11. ActiveMQ broker features in action

Consume from multiple destinations using wildcards

Session.createTopic("*.*.SubSubGroup");

Sending a message to multiple destinations

A composite destination uses a comma-separated name as the destination name.

session.createQueue("store.orders, topic://store.orders");

Advisory messages

Advisory messages are regular JMS messages generated on system-defined topics. They are a good alternative to JMX.

Virtual Topics

Virtual topics allow a publisher to send messages to a normal JMS topic while consumers receive messages from a normal queue. The topic name should follow the pattern: *VirtualTopic.<topic name>*. Virtual topics are a convenient mechanism to combine the load balancing and failover aspects of queues, with the durability of topics. If one of the consumer dies, the other consumer will continue to receive all messages on the queue.

Retroactive consumers

ActiveMQ has the ability to cache a configurable size of messages sent to a topic. The consumer needs to inform ActiveMQ it is interested in retroactive message, and the broker needs to be configured with the size of the cache.

Message delivery and dead-letter queues

When messages expire on the ActiveMQ broker (they exceed their time-to-live), or can't be delivered, they are moved to a dead-letter queue. A configurable POJO (Redelivery Policy) is associated with the connection that can be tuned to different policies).

There is one dead-letter queue for all messages, called ActiveMQ.DLQ. When a message is sent to a dead-letter queue, an advisory message is generated for it.

Extending functionality with interceptor plug-ins

ActiveMQ provides the ability to supply custom code to supplement broker functionalities. *Visualization* draws a diagram of all the connections and the destinations. Logging allows to log the messages sent or acknowledged on an ActiveMQ broker.

Apache Camel framework

Camel framework is a routing engine builder. It allows you to define your own rules, the sources, the destinations and how to process the messages. It extends the flexibility and functionality of ActiveMQ.

12. Advanced Client Options

Exclusive Consumers

ActiveMQ can be configured so that only one consumer listens to a queue, ensuring the order of arrival. If the consumer fails, another consumer will be activated.

Message Groups

Messages can be grouped together for a single consumer, by setting the message header JMSXGroupID.

ActiveMQ streams

Advanced feature to transfer very large file by breaking them into chunks.

Blob messages

A blob message does not contain the data being sent, but is a notification that a blob is available.

Scheduling messages to be delivered by ActiveMQ in the future

It is possible to schedule a message to be delivered after a delay, or at regular intervals. Messages to be sent are stored persistently.

13. Tuning ActiveMQ for performance

Persistence

Default delivery mode is persistent. Non-persistent messages are significantly faster than persistent messages (the producer does need to wait for a receipt from the broker).

```
MessageProducer producer = session.createProducer(topic);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

In ActiveMQ non-persistent delivery is reliable: the delivery of messages will survive network outages and system crashes, as long as the producer is active: it holds messages for redelivery in its failover transport cache.

Transactions

It is possible to batch up the production or the consumption of messages to improve performance.

```
Session session = connection.createSession(true, Session.SESSION_TRANSACTED);
producer.send(message);
session.commit();
```

Embedding brokers

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("vm://service");
```

Open Wire Protocol

Binary format used for transporting commands over a transport (such as TCP) to the broker.

Asynchronous send

Tell the `MessageProducer` not to except a receipt for messages it sends to the ActiveMQ broker.

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setUseAsyncSend(true);
```

Producer flow control

Producer flow control allows the message broker to slow the rate of messages that are passed through it when resources are running low.

Consumer

Some of the biggest performance gains are obtained by tuning the consumers.

ActiveMQ acknowledgment modes

```
Session.SESSION_TRANSACTED: Rolls up acknowledgments with Session.commit().
```

Session.CLIENT_ACKNOWLEDGE All: All messages up to when a message is acknowledged are consumed.

Session.AUTO_ACKNOWLEDGE: Automatically sends a message acknowledgment back to the ActiveMQ broker for every message consumed.

Session.DUPS_OK_ACKNOWLEDGE: Allows the consumer to send one acknowledgment back to the ActiveMQ broker for a range of messages consumed.

ActiveMQSession.INDIVIDUAL_ACKNOWLEDGE: Sends one acknowledgment for every message consumed.

14. Administering and monitoring ActiveMQ

JMX Agent: used to expose the ActiveMQ MBeans. The variable SUNJMX holds the JMX properties recognized by the JVM.

```
SUNJMX="-Dcom.sun.management.jmxremote"
```

The JMX agent in the JVM is controlled by the `com.sun.management.jmxremote` property, whereas the ActiveMQ domain is controlled by the `useJmx` attribute in the broker configuration file.

Creates connection to MBean server:

```
JMXServiceURL url = ...
JMXConnector connector = ...
MBeanServerConnection connection = connector.getMBeanServerConnection();
ObjectName name = new ObjectName("my-broker:BrokerName=localhost,
Type=Broker");
```

Queries for broker MBean:

```
BrokerViewMBean mbean = (BrokerViewMBean)
MBeanServerInvocationHandler.newProxyInstance(connection, name,
BrokerViewMBean.class, true);

System.out.println("Statistics for broker " + mbean.getBrokerId() + " - " +
mbean.getBrokerName());
```

MBean name:

```
<jmx domain name>:BrokerName=<name of the broker>,Type=Broker
```

Tools for ActiveMQ administration

`Bin/activemq`: start the broker

`Bin/activemq-admin`: to monitor the broker state from the command line

Command agent: allows you to issue administration commands to the broker using plain JMS messages.

`JConsole`: client application that allows you to browse and call methods of exposed MBeans.

Web Console: http://localhost:8161/admin (for dev environments)

`Broker loggin`: `cf. Data/activemq.log`